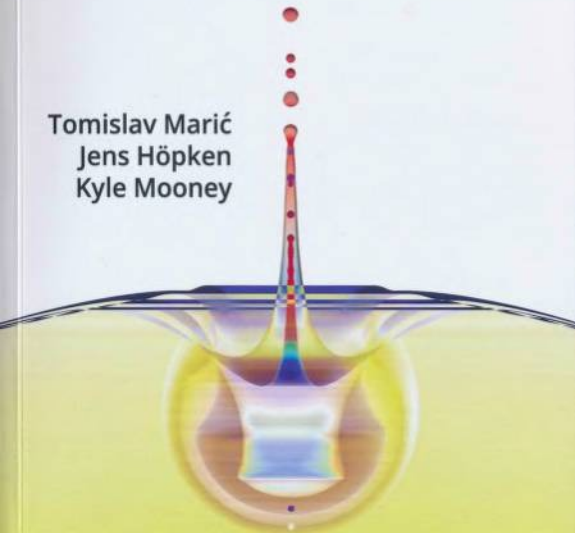


The **OpenFOAM**[®] Technology **Primer**

Tomislav Marić
Jens Höpken
Kyle Mooney



Published by sourceflux UG (haftungsbeschränkt)

Information on this title on www.sourceflux.de/book

© Tomislav Marić, Jens Höpken and Kyle Mooney. All rights reserved.

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of sourceflux.

OpenFOAM® and OpenCFD® are registered trademarks of OpenCFD Limited, the producer OpenFOAM software. All registered trademarks are property of their respective owners. This offering is not approved or endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks. Tomislav Marić, Jens Höpken, Kyle Mooney and sourceflux are not associated to OpenCFD. All product names mentioned herein are the trademarks or registered trademarks of their respective owners.

First published in print format in 2014

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

© Tomislav Marić, Jens Höpken and Kyle Mooney. All rights reserved.

ISBN 978-3-00-046757-8

Acknowledgments

First of all, we would like to thank our doctoral thesis supervisors for giving us the opportunity to undertake this endeavor, namely Dr. rer. nat. Dieter Bothe of the Technical University of Darmstadt, Prof. Dr.-Ing. Bettar Ould el Moctar of the University of Duisburg-Essen and Prof. David P. Schmidt of UMass Amherst.

Of course, this book would not have been possible without a group of competent reviewers, whose insights and suggestions have been very valuable. Thanks to Bernhard Gschaider, Michael Wild, Maija Benitz, Fiorenzo Ambrosino and Thomas Zeiss. In addition to the main reviewers, we have asked various friends and colleagues to proofread and provide comments and suggestions for different chapters. Thanks to Udo Lantermann, Matthias Tenzer, Andreas Peters and Irenäus Wlokas for going through the process of reading the first drafts of various chapters and proposing a great amount of changes.

Furthermore, we want to thank persons that have supported us in different ways and kept us motivated, as we wrote this book. These people are namely: Olly Connelly¹, Hrvoje Jasak, Iago Fernández, Manuel Lopez Quiroga-Teixeiro, Rainer Kaiser and Franjo Juretić.

¹<http://vpsBible.com>

To Marija, Kathi, Olivia and our families.

Table of Contents

Preface	1
What is covered in this book	2
Prerequisites	4
Intended audience	4
Conventions	5
 I Using OpenFOAM	 7
1 Computational Fluid Dynamics in OpenFOAM	9
1.1 Understanding The Flow Problem	9
1.2 Stages of a Computational Fluid Dynamics (CFD) Analysis	11
1.2.1 Problem Definition	11
1.2.2 Mathematical Modeling	11
1.2.3 Pre-processing and Mesh Generation	12
1.2.4 Solving	13
1.2.5 Post-Processing	13
1.2.6 Discussion and Verification	14
1.3 Introducing the Finite Volume Method in OpenFOAM . .	14
1.3.1 Domain Discretization	16
1.3.2 Equation Discretization	25
1.3.3 Boundary Conditions	30
1.3.4 Solving the System of Algebraic Equations	32
1.4 Overview of the Organization of the OpenFOAM Toolkit .	34
1.5 Summary	36
 2 Geometry Definition, Meshing and Mesh Conversion	 39
2.1 Geometry Definition	40
2.1.1 CAD Geometry	47
2.2 Mesh Generation	48
2.2.1 blockMesh	48

2.2.2	snappyHexMesh	57
2.2.3	cfMesh	67
2.3	Mesh Conversion from other Sources	78
2.3.1	Conversion from Thirdparty Meshing Packages	78
2.3.2	Converting from 2D to Axisymmetric Meshes	81
2.4	Mesh Utilities in OpenFOAM	85
2.4.1	Refining the Mesh by a Specified Criterion	85
2.4.2	transformPoints	87
2.4.3	mirrorMesh	88
2.5	Summary	90
3	OpenFOAM Case Setup	91
3.1	The OpenFOAM Case Structure	91
3.2	Boundary Conditions and Initial Conditions	95
3.2.1	Setting Boundary Conditions	97
3.2.2	Setting Initial Conditions	100
3.3	Discretization Schemes and Solver Control	104
3.3.1	Numerical Schemes (fvSchemes)	104
3.3.2	Solver Control (fvSolution)	117
3.4	Solver Execution and Run Control	120
3.4.1	controlDict Configuration	121
3.4.2	Decomposition and Parallel Execution	122
3.5	Summary	125
4	Post-Processing, Visualization and Data Sampling	127
4.1	Post-processing	127
4.2	Data Sampling	135
4.2.1	Sampling along a Line	136
4.2.2	Sampling on a Plane	138
4.2.3	Generating and Interpolating to an Iso-surface	140
4.2.4	Boundary Patch Sampling	141
4.2.5	Sampling Multiple Sets and Surfaces	143
4.3	Visualization	143
II	Programming with OpenFOAM	151
5	Design Overview of the OpenFOAM Library	153
5.1	Generating Local Documentation using Doxygen	155



5.2	Parts of OpenFOAM encountered during Simulations . . .	156
5.2.1	Applications	157
5.2.2	Configuration System	159
5.2.3	Boundary Conditions	160
5.2.4	Numerical Operations	160
5.2.5	Post-processing	165
5.3	Often Encountered Classes	167
5.3.1	Dictionary	167
5.3.2	Dimensioned Types	169
5.3.3	Smart Pointers	174
5.3.4	Volume Fields	188
6	Productive Programming with OpenFOAM	193
6.1	Code Organization	194
6.1.1	Directory Organization	196
6.1.2	Automating Installation	197
6.1.3	Documenting Code using Doxygen	201
6.2	Debugging and Profiling	202
6.2.1	Debugging with GNU Debugger (gdb)	202
6.2.2	Profiling with valgrind	207
6.3	Using git to Track an OpenFOAM Project	210
6.4	Installing OpenFOAM on an HPC cluster	213
6.4.1	Distributed Memory Computing Systems	213
6.4.2	Compiler Configuration	214
6.4.3	MPI Configuration	215
7	Turbulence Modeling	219
7.1	Introduction	219
7.1.1	Wall Functions	221
7.2	Pre- and Post-processing and Boundary Conditions	223
7.2.1	Pre-processing	224
7.2.2	Post-processing	225
7.3	Class Design	226
8	Writing Pre- and Post-processing Applications	229
8.1	Code Generation Scripts	230
8.2	Custom Pre-processing Applications	232
8.2.1	Decomposing and Starting a Parallel Run	232
8.2.2	Parameter Variation using PyFoam	235

8.3	Custom Post-processing Applications	240
9	Solver Customization	259
9.1	Solver Design	259
9.1.1	Fields	263
9.1.2	Solution Algorithm	264
9.2	Customizing the Solver	265
9.2.1	Working with Dictionaries	266
9.2.2	The Object Registry and regIOobjects	268
9.3	Implementing a new PDE	271
9.3.1	Additional Model Equation	271
9.3.2	Preparing the Solver for Modification	273
9.3.3	Adding new Entries into createFields.H	274
9.3.4	Programming the Model Equation	274
9.3.5	Setting up the Case	276
9.3.6	Executing the solver	279
10	Boundary Conditions	283
10.1	Numerical Background of a Boundary Condition	283
10.2	Boundary Condition Design	284
10.2.1	Internal, Boundary and Geometric Fields	284
10.2.2	Boundary Conditions	291
10.3	Implementing a new Boundary Condition	298
10.3.1	Recirculation Control Boundary Condition	299
10.3.2	Mesh Motion Boundary Condition	317
11	Transport Models	333
11.1	Numerical Background	333
11.2	Software Design	335
11.3	Implementation of a new Viscosity Model	342
11.3.1	Example Case	344
12	Function Objects	349
12.1	Software Design	350
12.1.1	Function Objects in C++	351
12.1.2	Function Objects in OpenFOAM	356
12.2	Using OpenFOAM Function Objects	360
12.2.1	Function Objects in the Official Release	360
12.2.2	Function Objects in swak4foam	362



12.3	Implementation of a Custom Function Object	367
12.3.1	Function Object Generator	367
12.3.2	Implementing the Function Object	370
13	Dynamic Mesh Operations in OpenFOAM	379
13.1	Software Design	381
13.1.1	Mesh Motion	381
13.1.2	Topological Changes	391
13.2	Usage	396
13.2.1	Global Mesh Motion	397
13.2.2	Mesh Deformation	398
13.3	Development	400
13.3.1	Adding Dynamic Mesh to a Solver	400
13.3.2	Combining two Dynamic Mesh Classes	404
13.4	Summary	427
14	Outlook	429
14.1	Numerical Methodology	429
14.2	Coupling to External Simulation Platforms	430
14.3	Workflow Enhancement	431
14.3.1	Graphical User Interfaces	431
14.3.2	Console Interfaces	432
14.4	Unconstrained Mesh Motion	432
14.4.1	Immersed Boundary Method	432
14.4.2	Overset Grid	433
14.5	Summary	434



List of Acronyms

VoF	Volume-of-Fluid
GPL	General Public License
DNS	Direct Numerical Simulations
CFD	Computational Fluid Dynamics
CAD	Computational Aided Design
FVM	Finite Volume Method
FEM	Finite Element Method
STL	Stereolithography
VTK	Visualization Toolkit
PDE	Partial Differential Equation
CV	Control Volume
r.h.s.	right hand side
RANSE	Reynolds Averaged Navier Stokes Equations
RANS	Reynolds Averaged Navier Stokes
RAS	Reynolds Averaged Simulation
LES	Large Eddy Simulation
DES	Detatched Eddy Simulation
DNS	Direct Numerical Simulation
CDS	Central Differencing Scheme
DSL	domain specific language
IPC	interprocess communication
MPI	message passing interface
RTS	Runtime Selection
GUI	Graphical User Interface
FSI	Fluid Structure Interaction
DSL	Domain Specific Language
RVO	Return Value Optimization
RAII	Resource Acquisition Is Initialization
HTML	HyperText Markup Language
IDE	Integrated Development Environment

UML	Unified Modeling Language
IO	Input/Output
HPC	High Performance Computing
VCS	Version Control System
OOD	Object Oriented Design
PISO	Pressure-Implicit with Splitting of Operators
SIMPLE	Semi-Implicit Method for Pressure-Linked Equations
AMI	Arbitrary Mesh Interface
SRP	Single Responsibility Principle
RBF	Radial Basis Function
IDW	Inverse Distance Weighted
SMP	Symmetric Multiprocessor
DMP	Distributed Memory Parallel



Preface

The use of the OpenFOAM toolkit for Computational Fluid Dynamics (CFD) is widely spread across industrial and academic environments. Compared to using proprietary CFD codes, the advantage of using OpenFOAM lies in the *Open-Source* General Public License (GPL) licensing which allows the user to *freely use* and *freely modify* a modern high-end CFD code. Such a licensing approach is of the highest importance to the users because it enables the toolkit to be used as a common platform for collaborative projects. Also, the development and implementation of novel or experimental methods is accelerated because the developer starts working from an existing code base and not from scratch. Engineers in industry see immediate cost reductions when moving their computational work to open source platforms due to the elimination of licensing fees.

In addition to the mentioned advantages, a disadvantage of using OpenFOAM is the larger amount of effort required to learn how to use and extend the platform, compared to many black-box commercial CFD platforms. Working with OpenFOAM (one might say this for Computational Science in general) requires a combination of knowledge stemming from different backgrounds, depending on the user's goal: software development and the C++ programming language, CFD, numerical mathematics, parallel programming and the general knowledge of physics. In order to work with or extend the available functionality in a sustainable way the user needs to understand the underlying numerics as well as software infrastructure of the toolkit. This book is an effort to describe the different facets of OpenFOAM in a single place, thus providing a starting point for the beginner user, and act as a reference for an intermediate or experienced user. The experienced user may consider this as a cookbook for performing specific tasks with OpenFOAM. We advise the beginner user to read the book from the beginning to the end and try to work on the provided examples on his/her own. This book covers two main aspects

of working with OpenFOAM: using the applications and developing and extending the library code.

In the first part of the book, we chose a few utilities and applications to describe the OpenFOAM work flow. This information should provide a sufficient starting point for the reader, who can investigate his/her interests further by following the provided instructions in a similar way for another solver or application. We have also provided an often missed general overview of the interplay between different toolkit elements involved in a CFD simulation using OpenFOAM in hope to produce a top-view perspective of this complex CFD software to the reader.

The second part of the book describes how one can program with the OpenFOAM library in a sustainable way using often encountered programming examples. We have tried to present the numerical background and software design of the specific programming example, together with its solution, hoping that the deeper understanding of the example will prepare the reader for other future programming tasks he or she may encounter.

What is covered in this book

Chapter 1 provides a general overview of the workflow involved with CFD simulations using OpenFOAM. A basic introduction of the Finite Volume Method (FVM) supported within OpenFOAM is provided with references pointing the reader to further information sources on this topic. An overview of the toolkit organization is presented as well as the interaction of the organizational elements within the scope of a CFD simulation.

Chapter 2 covers domain decomposition and discretization. This includes defining a geometry, mesh generation, and mesh conversion.

Chapter 3 describes the structure and the setup of a simulation case. This involves setting the initial and boundary conditions; configuring the run control parameters of a simulation, and numerical solver settings.

Chapter 4 gives an overview of the utilities used for pre- and post-processing as well as instructions on how to visualize the resulting data of a simulation.



- Chapter 5** provides a more detailed overview of the library than the one presented in chapter 1. In this chapter the reader will learn how to browse the code and where to find the building blocks of the library.
- Chapter 6** describes how to program with OpenFOAM in a productive and sustainable way. This chapter will be important for readers interested in programming with OpenFOAM who may lack a software development background. This chapter covers the development of self-sustained libraries, a way of using the git version control system, debugging and profiling, and so on.
- Chapter 7** introduces turbulence modeling into a simulation case. This involves setting up a turbulence model and its parameters.
- Chapter 8** is a first chapter that involves programming from the reader's side. Here we show how to develop pre- and post-processing applications using both C++ applications as well as commonly used utilities available for this purpose.
- Chapter 9** describes the background of the solver design in OpenFOAM, and shows how to extend an existing solver with new functionality.
- Chapter 10** shows the numerical background and software design aspects of boundary conditions in OpenFOAM. An implementation example of a custom boundary condition is provided that uses the principles described in chapter 6. As a result, the reader will develop a library of boundary conditions which is dynamically linked to the client code (a solver application).
- Chapter 11** covers the numerical background, design and implementation of transport models. As an example, an implementation of a temperature dependent viscosity model is provided.
- Chapter 12** introduces the use of function objects within OpenFOAM. The background of function objects in C++ is provided, as well as a list of references for further study. The implementation of function objects in OpenFOAM is described, in addition to an instructional programming example.
- Chapter 13** shows how to extend a solver with the functionality of the dynamic mesh in OpenFOAM. The available dynamic mesh engine in OpenFOAM is very powerful and enables the user to build his or her own dynamic mesh objects by agglomerating exiting ones.
- Chapter 14** gives an outlook of further advanced usage and programming topics with OpenFOAM.

Prerequisites

In order to compile and run the examples in this book you'll require access to a computer running a relatively up-to-date Linux distribution. While working with OpenFOAM you will be spending a lot of time interacting with the Linux terminal so you should be capable of maneuvering the directory structure, running executables, and editing text files. All command line examples illustrated will use the Bash Unix shell. A detailed description on how to install OpenFOAM can be found on <http://www.openfoam.org/download/source.php>. The contents of this book have been developed using version 2.1.1.

We assume that you possess a reasonable familiarity with the C++ programming language and general paradigms associated with the concept of Object Oriented Design (OOD) and generic programming. Additionally, it is not possible to learn and understand OpenFOAM without knowing the basics of Computational Fluid Dynamics. If you would like to catch up on the above mentioned topics, have a look at the literature suggestions at the end of this chapter.

Whenever an expression is encountered that requires a higher level of background knowledge, it is suggested to follow up on the additional literature and information available on the internet. OpenFOAM is built upon software development, C++ programming language, numerical mathematics, fluid dynamics, etc. Covering all those aspects in detail in a single book is not possible.

Intended audience

Irrespective of your background with regards to CFD and OpenFOAM, you will gain additional knowledge with regards to OpenFOAM from this book. If you are just starting to dive into the world of CFD and Open Source software but have gained knowledge of fluid mechanics during your studies, Part 1 should act as an introduction into these new fields.

Even if you already are an experienced user of other CFD tools, you can skip the first two sections and quickly skim over the rest of the book's first part to get a grasp on working with OpenFOAM. The second part



of the book will give you guidance while you start to change the existing code base to suit your specific needs.

It should be noted that this book is not meant to be a substitute for the detailed texts on Numerical Analysis, Advanced Fluid Mechanics, or Computational Fluids Dynamics currently available.

Conventions

There are naming and typesetting conventions which will be used consistently throughout the book and should be illustrated before continuing. Firstly anything that must be entered on the command line is typeset using typewriter with a prepending `?>` . An example is shown below:

```
?> ls $FOAM_TUTORIALS
Allclean basic          electromagnetic lagrangian
Allrun  combustion    financial      mesh
Alltest compressible  heatTransfer  multiphase
DNS     discreteMethods incompressible  stressAnalysis
```

A code block of C++ is set as follows:

```
template<class GeoMesh>
tmp<DimensionedField<scalar, GeoMesh> > stabilise
(
    const DimensionedField<scalar, GeoMesh>&,
    const dimensioned<scalar>&
);
```

As it will be shown later, almost all definitions for an OpenFOAM simulation are done in various dictionaries. These dictionaries are plain text files, that can be accessed with any editor. If we want to show parts of such a dictionary, it is highlighted reading:

```
ddtSchemes
{
    default      Euler;
}
```

With this book covering various mathematical aspects of OpenFOAM, we will provide various equations that must be explained. For these equations, vectorial properties are typeset in bold face (e.g. velocity **U**).

Scalar properties, such as the flux ϕ employ normal face and tensors use bold face and are underlined (e.g. unit matrix **E**).

The example code and example cases of this book are available online, on github and bitbucket. Links to these repositories can be found on sourceflux.de/book. We are trying to keep them up-to-date and fix bugs as they occur. There will be difference between the code printed in the book and the one in the repository, though. This is mainly due to the limited space of this book. In order to keep things centralized, the errata of the book is located on sourceflux.de/book/errata.

Further reading

- Ferziger, J. H. and M. Perić (2002). *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer.
- Kundu, Pijush K., Ira M. Cohen, and David R Dowling (2011). *Fluid Mechanics with Multimedia DVD, Fifth Edition*. 5th ed. Academic Press.
- Lafore, Robert (1996). *C++ Interactive Course*. Waite Group Press.
- Lippman, Stanley B., Josée Lajoie, and Barbara E. Moo (2005). *C++ Primer (4th Edition)*. Addison-Wesley Professional.
- Newham, Cameron and Bill Rosenblatt (2005). *Learning the bash shell*. O'Reilly.
- Stroustrup, Bjarne (2000). *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Versteeg, H. K. and W. Malalasekera (1996). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method Approach*. Prentice Hall.



Part I

Using OpenFOAM

1

Computational Fluid Dynamics in OpenFOAM

This chapter helps getting started with approaching a CFD problem. The very basics of the Finite Volume Method (FVM) are covered and the top-level structure of the OpenFOAM platform is described.

1.1 Understanding The Flow Problem

The goal of any CFD analysis is to obtain a deeper understanding of the problem under consideration. As simulation results are often accompanied by experimental data, the validity of results and the overall aim of the simulation need to be taken into account. Furthermore, the simulated thermo-physical phenomenon needs to be properly mathematically modeled. Which in turn requires the CFD engineer to make a valid choice of the appropriate simulation solver application within the OpenFOAM framework. Additionally, engineering assumptions are made, that may complicate or simplify the CFD analysis. Analysis and reduction of the simulation domain geometry is often performed, taking into consideration the available computational and other resources.

Pragmatic questions that might be posed before undertaking a CFD analysis project are outlined below. This list is by no means exhaustive.

General considerations

- What is to be the conclusion resulting from the CFD analysis?
- How is the degree of accuracy of the results defined?
- With what methods will the results be validated?
- How much time is available for the project?

Thermo-physics

- Is the flow laminar, turbulent, or transitional?
- Is the flow compressible or incompressible?
- Does the flow involve multiple fluid phases or chemical species?
- Does heat transfer play an important role in the problem?
- Are the material properties functions of dependent variables? For example: temperature dependent viscosity, shear thinning fluid, etc.
- Is there sufficient information available regarding the boundary conditions, are they appropriately modeled and approximated?

Geometry and mesh

- Can an accurate discrete representation of the flow domain be constructed?
- Will the computational domain be deforming or moving during the simulation?
- Where can the complexity of the domain be reduced without impacting the solution accuracy?

Computational Resources

- How much computational time is available for the simulation?
- What kind of distributed computing resources are available?
- Will one simulation suffice or is a multi-run CFD analysis necessary?

These categorized questions contribute to a complete and proper CFD analysis of any flow problem. Skills in using OpenFOAM or any commercial fluid simulation package are rendered moot without a proper understanding of physics, numerical methods, as well as available computational resources. The interdisciplinary nature of CFD greatly contributes



to its complexity.

1.2 Stages of a CFD Analysis

An analysis based on CFD methods can usually be subdivided into 5 major components. Some of these steps must be performed multiple times in order to finally obtain results of the desired high quality.

1.2.1 Problem Definition

From the engineering point of view, the problem needs to attain the simplest possible form, that still accurately describes the actual real world engineering system. As a result, the important parameters of the simulated system are retained and the unimportant ones are neglected. For example, simulating a flow over an airfoil often involves considering the air to be an incompressible fluid.

1.2.2 Mathematical Modeling

After having defined the problem properly, it has to be formulated mathematically, with regards to the assumptions previously made. The mathematical model formulation is not performed by the CFD engineer, but by mathematicians and theoretical physicists. However, a CFD engineer must understand the models used to describe different physical phenomena. Within the OpenFOAM framework, the user has a choice between dozens of solver applications. Each application implements a specific mathematical model and choosing the right one is crucial in order to obtain a valid solution to the simulated problem.

Following on the flow over airfoil example, the incompressible assumption will exclude the solution of the energy equation. As another example, a potential flow is solely governed by Laplace's equation - details can be found in the book by Ferziger and Perić (2002). If more complex physical transport phenomena are taken into account, the complexity of the mathematical model increases. This usually leads to more sophisticated mathematical models, e.g. the Reynolds Averaged Navier Stokes Equations (RANSE), used to model the turbulent flow. The mathematical model describes the details of the flow, which means that the numeri-

cal simulation, that only approximates the solution of the model at best, cannot produce more information about the flow than described by the model itself. More information on turbulence modeling in OpenFOAM can be found in chapter 7. More details on the particular mathematical models can be found in fluid dynamics text books.

1.2.3 Pre-processing and Mesh Generation

The mathematical model defines fields of physical properties as dependent variables of the model equations. In CFD, more often than not, the equations describe a boundary and initial value problem. Therefore, the fields need to be initially set (pre-processed) before the start of the simulation. If the field values are spatially varying, different utility applications (utilities) may be used to compute and pre-process the fields. There are utilities that are distributed along with OpenFOAM (e.g. the `setFields` utility), or are a part of another project (e.g. `funkySetFields` utility of the `swak4Foam` project).

The use of some of the available pre-processing utilities is explained in chapter 8.



TIP

More information on the `swak4Foam` project can be found on the OpenFOAM wiki on <http://openfoamwiki.net/index.php/Contrib/swak4Foam>.

The flow domain must be discretized in order to approximate the model solution numerically. The spatial discretization of the simulation domain consists of separating the flow domain into a *computational mesh* consisted of volumes (cells), often of different shapes. All of these volumes taken together are referred to as the mesh or the computational grid. Usually, the mesh must be refined in areas of interest: for example in those parts of the domain where large gradients of field values occur. Further on, the accuracy and a proper choice of the mathematical model has to be kept in mind. Resolving flow features in a spatial manner does not compensate for a model that does not account for these features in the first place. On the other hand, increasing the mesh resolution for transient simulations might slow down the simulation This is due to the very



small value, set for the discrete time step in order to obtain a stable solution, when explicit discretization schemes are used.¹ The mesh is one of the most likely component of the simulation workflow that needs to be changed if the numerical simulation fails to converge. Failing simulations are very frequently caused by a mesh of insufficient quality. OpenFOAM comes with two different mesh generators which are namely `blockMesh` and `snappyHexMesh`. The usage of both is covered in chapter 2.

Additionally, pre-processing covers various other tasks, such as decomposing the computational domain if the simulations are run in parallel on multiple computers or CPU cores.

1.2.4 Solving

Alongside mesh generation, this usually is the most time consuming part of the CFD analysis. The time required highly depends on the mathematical model, the numerical scheme used to approximate its solution, as well as the geometrical and topological nature of the computational mesh. The differential (analytical) model gets replaced by a system of (linear) algebraic equations. In CFD, such algebraic linear equation systems are *large*, resulting with matrices with millions of coefficients. The algebraic equation systems are solved using algorithms developed specially for this purpose - iterative linear solvers. OpenFOAM framework supports a wide choice of linear solvers and although all the solver applications have pre-set choices of linear solvers and parameters. A skilled CFD engineer has hence the opportunity to modify both the solver and the corresponding parameters. This, in turn, may result in a much faster and/or accurate computation.

1.2.5 Post-Processing

After the simulation finished successfully, the user ends up having a large amount of data that must be analyzed and discussed. The data must be visualized appropriately in order to inspect the details of the flow. By using dedicated tools such as `paraView`, such data can be discussed fairly easily. For analyzing the simulation results, OpenFOAM provides

¹More information on this issue is provided in CFD textbooks, such as the book of Ferziger and Perić 2002.

a wide choice of post-processing applications. More details on various post-processing tools and methods are provided in chapter 8.

TIP

The standard post-processing tool, that comes with OpenFOAM is paraView. It is an open source tool and can be obtained freely from www.paraview.org.

1.2.6 Discussion and Verification

This is the point where the user has to determine whether to trust the results or not. The code solely does what the user instructs and with a large number of parameters, there is ample room for errors. If a mistake is made in one of the previous steps, it will most likely be discovered during the discussion.

When experimental data are available to compare the simulation results to, there is a more strict safety margin when it comes to confidence in the simulation results. If the simulation results do not satisfy the requirements, the previous steps of CFD analysis must be revisited.

1.3 Introducing the Finite Volume Method in OpenFOAM

This section provides a very brief overview of the FVM in OpenFOAM. The reader is directed to Ferziger and Perić (2002), Hrvoje Jasak, Jemcov, and Tuković (2007), Versteeg and Malalasekera (1996), and Weller, Tabor, H. Jasak, and Fureby (1998) for further details regarding this topic. It would be far beyond the scope of this book to describe the FVM from the bottom up.

WARNING

The unstructured mesh topology and the FVM may both be complex topics for a CFD novice. It might be worthwhile to return often to this part of the book if something is not understood later on.

Steps of the unstructured FVM in OpenFOAM correlate somewhat to the



steps of the CFD analysis described in section 1.2. The physical properties that define the fluid flow, such as pressure, velocity, or temperature are dependant variables in a *mathematical model*: a formal mathematical description of the fluid flow. A mathematical model that describes a fluid flow is defined as a system of Partial Differential Equations (PDEs). Seemingly different physical processes are sometimes described using a same mathematical description, e.g. the conduction of heat as well as diffusion of sugar concentration in water are modelled as *diffusive processes*. The general scalar transport equation, as described by Ferziger and Perić 2002, holds the terms used to model different physical processes, for example: transport of particles with the fluid velocity (advection term), heat source (source term), and so forth. As it contains often encountered terms, the general scalar transport equation can be used to describe the FVM, and it is given as

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{U}\phi) + \nabla \cdot (D\nabla\phi) = S_\phi, \quad (1.1)$$

With ϕ being the scalar property, \mathbf{U} the velocity vector and D the diffusion coefficient. The terms in equation (1.1) from left to right are: temporal term, convective term, diffusive term and source term. Each term describes a physical process that changes the property ϕ in a different way.

Depending on the nature of the process, some of the terms may be neglected: e.g. for the inviscid fluid flow, the diffusive term (transport) of the momentum is neglected. Hence this term is removed from the momentum equation. In addition, the coefficients that appear in some of the terms may be constant values, or spatially and/or temporally varying fields themselves. An example of such a coefficient is a temperature dependant conductivity coefficient for conductive heat transfer: $\nabla \cdot (k\nabla T)$, which makes k a spatially (and possibly temporally) varying field that would depend on the model solution, i.e. the temperature field.

The purpose of any numerical method is to obtain an approximation of a solution of the mathematical model. A numerical approximation of the solution of a complex physical process is necessary, since the exact solution can solely be obtained for some special cases, that often lack relevance for technical applications. An example for this is a Couette flow model, for which an analytical solution exists.

TIP

Couette flow is a flow between two plates of assumed infinite dimensions, where the top plate translates with a constant velocity. This flow problem will be picked up in the later chapters.

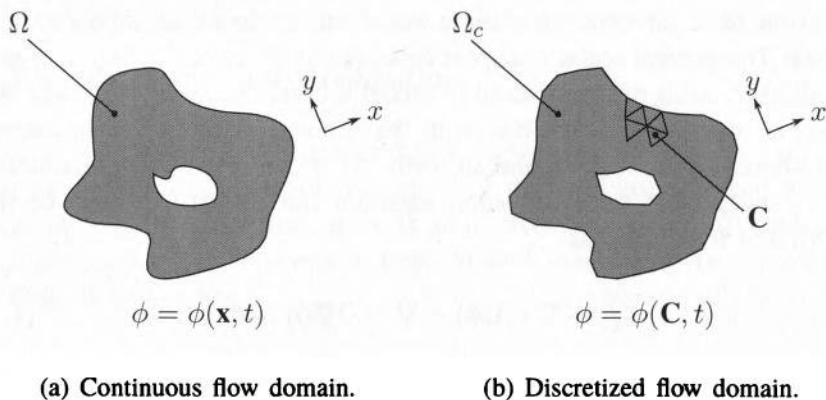


Figure 1.1: Continuous and discretized flow domains and the corresponding flow fields.

Usually, the solution of the mathematical model is obtained by solving a discrete approximation of the governing system of equations. The approximation process of a numerical method involves a substitution of the system of PDEs with a corresponding system of algebraic equations, that *can be solved*. These algebraic equations are evaluated at discrete points in space, e.g. at cell centers \mathbf{C} . The generation of an algebraic system of equations within the framework of the FVM is made up of two major steps: domain discretization and equation discretization.

1.3.1 Domain Discretization

As previously stated, the mathematical model uses continuous fields that describe the flow field at any point in space. In order to solve the equations of the mathematical model, this continuous space must be discretized into a finite number of volumes (cells). The finite volumes (cells) make up the finite volume mesh. The transition from the continuous rep-



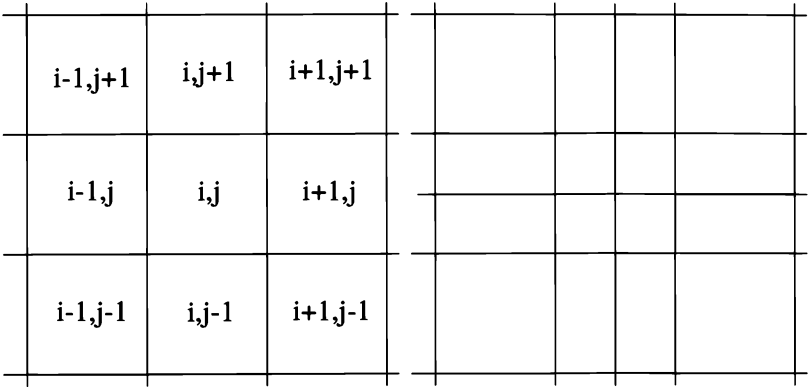
resentation of the flow with continuous fields filling the flow domain Ω , to a discretized domain Ω_D is shown in figure 1.1. Continuous fields that are defined in each point of the space filled by the fluid in figure 1.1a are replaced by discrete fields which stored in the centers of the finite volumes C , as shown in figure 1.1b. Each finite volume stores an averaged value of the physical property (e.g. temperature) in its cell centre C . Taking all of the cell centers together, the discrete field for the particular physical property can be assembled, e.g. the discrete cell-centred temperature field.

There are different ways of discretizing the domain, which result in topologically different *finite volume meshes*. Topologically different domain discretizations determine the shapes of the finite volumes and, consequently, how the components of the mesh (cells, faces, points) are connected with each other. More information on mesh generation can be found in chapter 2.

We distinguish between three major types of meshes: structured, block-structured and unstructured meshes. Please note that all three have different requirements to both domain discretization, equation discretization and the way the source code has to be formulated. The mesh topology and related access optimizations of the mesh may have a direct impact on the accuracy and efficiency of numerical operations performed by the numerical library as well as how the algorithms are parallelized, which is the case in OpenFOAM.

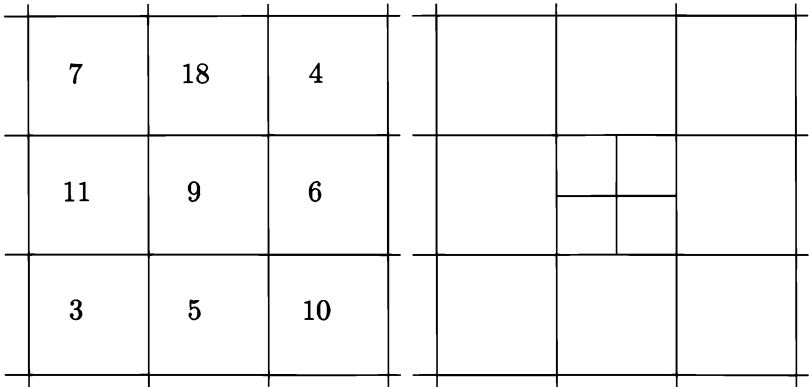
Structured meshes support direct addressing of an arbitrary cell neighbor as well as direct cell traversal: the cells are labeled with the indexes increasing in the directions of the coordinate axis (see figure 1.2a). Unstructured meshes on the other hand have no apparent direction (see figure 1.3a). Neither in the way the cells are addressed, nor their topology will be a result of a geometrical algorithm used to discretize the computational domain, which is un-ordered by nature.

A structured mesh topology increases the absolute accuracy of the interpolations involved in the FVM, but it makes the mesh less flexible when it is used for mesh generation of geometrically complex domains. During the mesh generation process, the user usually desires to generate a dense mesh where large gradients occur and not to waste cells in flow regions



(a) Sub-set of a 2D equilateral Cartesian mesh. (b) Refined sub-set of a 2D equilateral Cartesian mesh.

Figure 1.2: Structured quadratic mesh.



(a) Sub-set of a 2D unstructured quadratic mesh. (b) Refined sub-set of a 2D unstructured quadratic mesh.

Figure 1.3: Unstructured quadratic mesh.

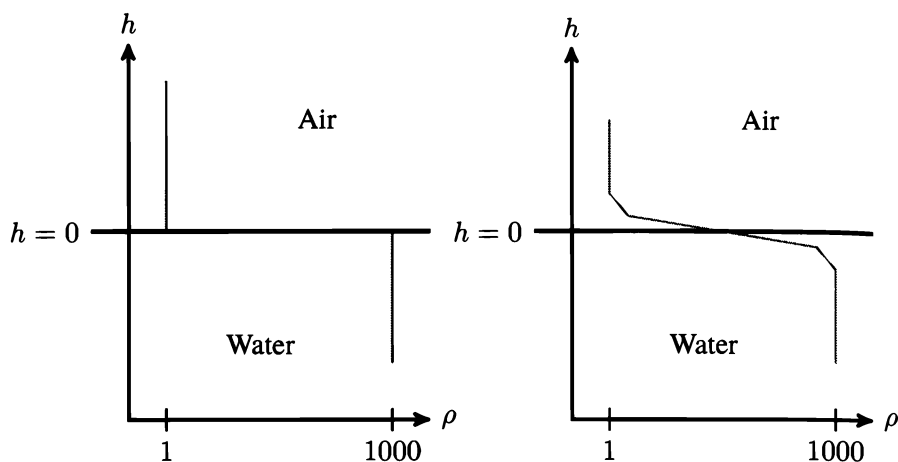


where no such gradients occur. All mesh generation steps should happen in the shortest possible time. This is impossible to achieve with structured meshes without substantial enhancements. Local mesh refinements are virtually impossible to achieve, since the refinement has to be propagated into the respective direction through the entire mesh. An example sketch of the difference in topology between the structured and unstructured mesh is given in figure 1.3b.

Figure 1.2a shows a two dimensional schematic of a structured quadratic mesh. This kind of mesh is also called Cartesian, since it is consisted of volumes with volume centers distributed in the direction of the coordinate system axes. Being able to move through the mesh by changing the indexes i, j has one significant advantage: the numerical method working with this kind of mesh may access any neighboring cell by simply incrementing or decrementing the indexes i, j by an integer value of 1. This comes in handy e.g. when the flux through the cell faces ϕ_f is interpolated from cell centers to the face centre: high-order interpolation stencils can be easily applied, to increase the accuracy of the solution. A high-order (large) interpolation stencil means that an increased number of cells, that need not necessarily be face-neighbors of the current cell, are included into the interpolation.

There is one problem, however: it is rather impossible to refine the mesh locally. Locally in this context means, in a subregion of the mesh. This is often the case in a region of extreme changes of the physical property, when the order of interpolation available on the structured mesh is still insufficiently accurate, to capture the large jump of a physical property. Such large jumps in the values of physical properties are present e.g. in two-phase flow simulations where two immiscible fluids are simulated. An interface is formed between two fluids that separates them, and the values of the physical properties may vary by orders of magnitude, as can be seen in a schematic diagram in figure 1.4. A good example for such a flow regime is a water-air two-phase system with a ratio of densities being approximately 1000 (Ubbink 1997 and Rusche 2002 deal with such simulations within the OpenFOAM framework).

To resolve such steep gradients in the fields, local mesh refinement is often applied. This refinement can either be done during pre-processing or applied adaptively during runtime. As mentioned previously, refining



(a) Continuous space with sudden jump of the density ρ (b) Discretized space with gradual - but still steep - jump to the density ρ

Figure 1.4: Qualitative distribution of the density ρ with respect to the height h over the free surface.

a structured mesh cannot be done locally: the topology of a structured mesh forces the mesh to be refined in the complete direction (see figure 1.2b), where the refinement of a single cell in two directions generates refinement throughout the entire mesh. Structured meshes that conform to curved geometries are especially difficult to generate. A mathematical parameterization of curved domain boundaries (coordinatization) is necessary in order to maintain the structured mesh topology.

There are, however, extensions of the structured mesh discretization practice and some of them allow for both local and dynamic mesh refinement. In order to increase the accuracy locally, block structured refinement may be used, which is a process of building a mesh that consists of multiple structured blocks. When such a block structured mesh is assembled, the blocks will have different local mesh densities. This introduces new requirements, though: The numerical method must either be able to deal with non-conforming block patches (hanging nodes). Or the block refinement needs to be carefully crafted, so that the points on adjoining blocks of different densities *match perfectly* (patch-conforming block-meshes).



Building block structured meshes is a complex problem even for simple flow domains, which makes block-structured meshes a poor choice for many technical applications, that involve complex geometries of the flow domain. Refining block-structured meshes results in refinement regions spreading through the blocks. With standard solvers that rely on patch-conforming block-meshes the refinement complicates the mesh generation even more.

Dynamic adaptive local refinement in OpenFOAM is performed by introducing additional data structures that generate and store the information related to the refinement process. An example of such method is an *octree based refinement*, where an octree datastructure is used to split the cells of the structured Cartesian mesh into octants. This requires the mesh to consist solely of hexahedral cells. Information stored by the octree data structure is then used by the numerical interpolation procedures (discrete differential operators) taking into account the topological changes resulting from local mesh refinement. The possibility of dealing with more complex geometrical domains can then be added to an octree-refined structured mesh by using an *cut-cell* approach. In that case, the cells which hold the curved domain boundary, are cut by a piecewise-linear approximation of the boundary. Octree-based adaptive mesh refinement may have an advantage in its efficiency depending on the way the topological operations are performed on the underlying structured mesh. However, the logic of the octree based refinement requires the initial domain to be box-shaped. More information about local adaptive mesh refinement procedure can be found in the book *Adaptive Mesh Refinement - Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3-5, 2003 (Lecture Notes in Computational Science and Engineering)* 2005.

OpenFOAM implements a FVM of second order of convergence with support for *arbitrary unstructured meshes*. Arbitrary unstructured means that, in addition to the unstructured mesh topology, the mesh cells can be of arbitrary shape. This allows the user to discretize flow domains of very high geometrical complexity. The unstructured mesh allows for a very fast, sometimes even *automatic mesh generation* procedure. Which is very important for industrial applications, where the time needed to obtain results is of great importance. A short introduction to the mesh generation is provided in chapter 2.

Figure 1.3a shows a two-dimensional schematic of a quadratic unstructured mesh. Since the mesh addressing is not structured, the cells have been labelled solely for the purpose of explaining the mesh topology. The unordered cells complicate the possibility to perform operations in a specific direction without executing costly additional searches and re-creating the directional information locally. Another advantage of the unstructured mesh is the ability for a cell to be refined locally and directly, which is sketched in figure 1.3b. The local refinement is more efficient in terms of increase of the overall mesh density, since it only increases the mesh density where it is required.

WARNING

Although the topology of the mesh is fully unstructured in OpenFOAM, often the mesh will be generated block-wise for cases with simple domain geometry (`blockMesh` utility): this does not mean that a *block structured* mesh is generated.

The next component to be described is how neighboring cell values can be addressed in the unstructured FVM, as it is implemented in OpenFOAM. This is an essential component, as each cell needs to communicate with its neighbors. It is important to distinguish between the basic principles of that communication and how this communication is actually implemented, in terms of mesh format. The first is explained in the following, whereas the way the mesh is stored and how this storage method helps with the communication, is addressed in chapter 2.

The way the mesh elements are addressed by the algorithms of the numerical method is determined by the mesh topology. OpenFOAM defines its topological mesh structure using: *indirect addressing*, *owner-neighbor addressing* and *boundary mesh addressing*.

Indirect addressing defines how the mesh is assembled from the mesh points, which are defined explicitly as a list of points. The faces are defined by the mesh points, but rather than using the point coordinates for that, the position of the particular point in the list of points is used. Similarly, the cells are constructed from the faces, by referring to the position of the face in the faces list (see figure 1.5). Indirect addressing avoids copying of mesh points whenever an



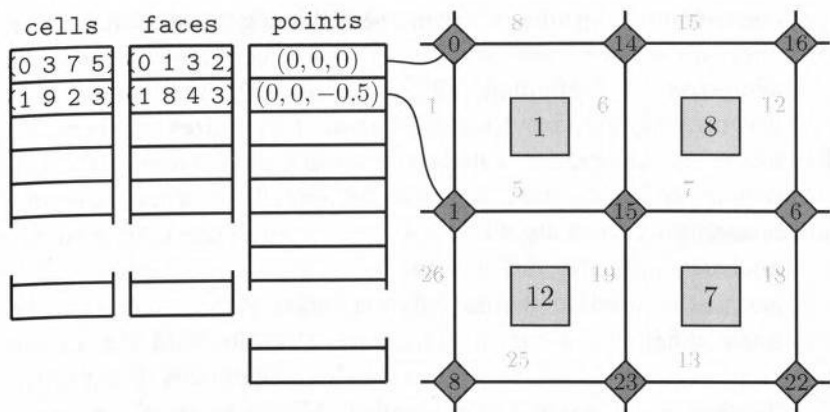


Figure 1.5: Excerpt of a mesh, with labels for cells (squares), points (diamonds) and faces. The labels and coordinates are just for representational purposes and the view is reduced to 2D.

instance of a face or cell is created. Otherwise one would end up having multiple copies of the same points and faces in memory, which would be a waste of computing capacity and would severely complicate topological operations.

Face 1, as shown in figure 1.5, serves as an example. It is constructed from points 0 and 1, as well as 3 and 2, which are not shown in figure 1.5. This face in turn is used to assemble cell 1.

Owner-neighbor addressing defines which cell owns a certain face and which is its neighbour. In addition it is an optimization which defines the way the indexes in the mesh faces are ordered. It sets the direction of the face area normal vector \mathbf{S}_f shown as an arrow in figure 1.6. Two global lists are introduced into the mesh with owner-neighbor addressing optimization: the *face-owner* and the *face-neighbor* list. For each face of the mesh, there may be only two adjacent cells. With one cell being the face-owner cell (marked with P in figure 1.6) and the other a face-neighbor (marked with N in figure 1.6). The owner cell of a face will be the cell with a lower index in the list of mesh cells. This information determines the ordering of the face indexes: the face area normal vector is directed always from the owner into the neighbor cell. Switching the orientation of the face area normal is an efficiency optimization which is done to reduce redundant computations in the equation

discretization step described in the following subsection.

Boundary addressing takes care of how the faces on a boundary are addressed. By definition, all faces that only have an owner cell and no neighbor, are boundary faces. It optimizes the face access efficiency: isolate the boundary faces and store them at the end of the list of mesh faces. This allows an efficient definition of the boundary patches as subsets of the mesh face list. In turn, the boundary mesh is defined as a set of boundary patches, which are finally used to define different physical boundary conditions. Such definition of the boundary mesh results with the automatic parallelization of all the top-level code in OpenFOAM that relies on the face-based interpolation practice. All the faces of the boundary mesh are directed outwards from the flow domain which means that they have only one cell owner.

TIP

This section is aimed to describe the mesh composition, from a CFD point of view. More specifics about how the mesh is actually composed, are given in chapter 2.1.

While indirect addressing, as well as the rest of the unstructured mesh features, increase the flexibility of the numerical simulation code, the indirection decreases the performance when compared with meshes that could also be handled by block-structured code.

WARNING

The computational efficiency is often misjudged when the numerical simulation frameworks, that work with direct addressing on Cartesian meshes are compared with OpenFOAM. The comparison is often done for two block meshes with the claim that the meshes are "the same" - the meshes are indeed obviously very different.

A schematic sketch of how cell-centred values of the unstructured mesh are addressed by the face owner-neighbor addressing mechanism is shown in figure 1.6. The example cell has the cell label 0 and it shares three faces with neighboring cells, and one boundary face b . This cell has the lowest index of all its neighbors (2,3,4), which makes it the owner of all



its faces - it takes on the P index for each of its faces. This results in the face area normal vectors being oriented outward from the cell 1 to all of its neighbors in this example.

Additional addressing, such as cell-cells and point-cells, is also stored by the unstructured mesh in OpenFOAM. The additional addressing can be used to construct more specialized methods, e.g a specialized interpolation scheme. The actual indexing of all the connectivity is a direct result of the mesh generation algorithm.

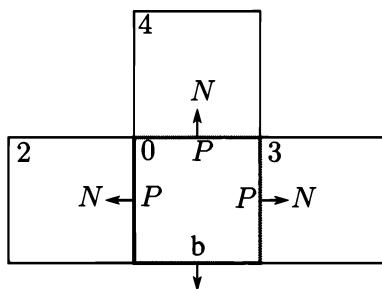


Figure 1.6: Owner-neighbor addressing for an example cell with index 0 and blue frame. A label pair (N, P) is defined for each cell face. The face-adjacent cell that has the lower cell label (upper left corner) is the owner cell P , and b marks the boundary face of cell 0.

1.3.2 Equation Discretization

Once the domain is discretized into finite volumes, approximations are applied to the terms of the mathematical model. Transferring the differential terms into discrete differential operators. In contrast to the domain discretization, this is done during runtime in each solver in OpenFOAM. An exception are solvers that employ dynamic meshes, that may perform the domain discretization repeatedly. Detailed descriptions of the equation discretization in OpenFOAM are provided by Jasak 1996, Ubbink 1997, Rusche 2002 and Juretić 2004, among others. Only the discretization of a simple advection equation for a scalar property ϕ with the velocity \mathbf{U} without source terms, is described in the following:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{U}\phi) = 0 \quad (1.2)$$

Equation (1.2) has two terms: the temporal term and the advective term. Both terms need to be discretized in order to obtain the algebraic equation, since the equation cannot be solved in the existing form analytically. The numerical method must be consistent (see Ferziger and Perić 2002): as the size of the cells is reduced, the discrete (algebraic) mathematical model must approach the exact mathematical model. Or in other words: as described by Ferziger and Perić (2002), refining the computational domain infinitely and solving the discretized model on this spatial discretization leads to the solution of the mathematical model consisting of PDEs. To obtain the discrete model, equation (1.2) is integrated in time and space:

$$\int_t^{t+\Delta t} \int_{V_P} \left(\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{U}\phi) \right) dx dt = 0. \quad (1.3)$$

Integration of the temporal term of equation (1.2) can be approximated as

$$\int_t^{t+\Delta t} \int_{V_P} \frac{\partial \phi}{\partial t} dx dt \approx V_P \frac{\phi^n - \phi^o}{\Delta t}, \quad (1.4)$$

where V_P is the cell volume, n and o denote the new and the old time step of the simulation respectively, and Δt denotes the time step value. The time step is introduced since time is to be discretized as well into a sequential finite intervals (time steps).

The advective term is discretized by integrating and applying the Gauss divergence theorem:

$$\int_t^{t+\Delta t} \int_{V_P} \nabla \cdot (\mathbf{U}\phi) dx dt = \int_t^{t+\Delta t} \int_{\partial V} \phi \mathbf{n} do dt \approx \sum_f \phi_f \mathbf{U}_f \mathbf{S}, \quad (1.5)$$

Where ∂V marks the continuous boundary of a finite volume V_P (a surface bounding the volume V_P) with the area differential do , f marks a face of a cell, and \mathbf{S} marks the outward-pointing face area normal vector. The face area normal vector is a vector normal to the cell face with the magnitude of the face area. Some of the values stored in the face centre must somehow be known on the face, in order to proceed with the computation of the right hand side (r.h.s.) of equation (1.5). This is done using interpolation and the interpolated values are marked by the index f , e.g. ϕ_f . Specifics about this interpolation are provided at the end of this chapter. Considering two discrete algebraic terms, equation (1.2)



takes the following discrete form:

$$V_P \frac{\phi^n - \phi^o}{\Delta t} + \sum_f \phi_f \mathbf{U}_f \mathbf{S} = 0. \quad (1.6)$$

It is easy to see that in the limit where both Δt and V_P tend to zero, the discrete equation (1.6) corresponds to the continuous mathematical model shown in equation (1.2). No indexes marking the new (n) or the old (o) time step are present in equation (1.5). This is a result of neglecting variations of the face interpolated values in time.

Depending on the choice of the new or the old time step for the final term of equation (1.5), the resulting algebraic equation will be solved *explicitly* or *implicitly*. The differences are presented in the following:

Explicit temporal discretization: If the spatial terms are evaluated in the old time step, the only value from the new time step is the value stored in the centre of the Control Volume (CV). For this, the algebraic equation is assembled:

$$V_P \frac{\phi^n - \phi^o}{\Delta t} + \sum_f \phi_f^o \mathbf{U}_f^o \mathbf{S} = 0, \quad (1.7)$$

And in this case, we can simply put all the old values on the r.h.s. of the equation and compute the new cell value ϕ^n by *explicitly* evaluating the r.h.s. of the equation.

Implicit temporal discretization: In case of evaluating the evaluating the r.h.s. in the new time step:

$$V_P \frac{\phi^n - \phi^o}{\Delta t} + \sum_f \phi_f^n \mathbf{U}_f^n \mathbf{S} = 0, \quad (1.8)$$

The algebraic equation assembled for the cell in question will carry dependant variables from the surrounding cells in the new time step. Which means that such an equation must be assembled, for each CV to construct the system of algebraic equations and solve that system to get the entire cell-centred field in the new time step. This kind of solution is called an *implicit solution*.

The coefficients of the discrete equation will be determined by the following factors:

- the way cell centred values are interpolated to the faces (U_f and ϕ_f) using different interpolation methods,
- the geometrical shape of the cell (especially the number of cell faces) as well as the number and geometry of adjacent cells. The cell shape determines the position of the cell centre, and thus has an impact on the interpolation,
- the size of the cell: the smaller the V_P , the closer the algebraic equation will be to the exact equation, hence increasing the accuracy of the solution,
- what terms are present in the equation: a diffusive term and/or a source term may be added, whose discretization will change the values of the coefficients in the final algebraic equation,
- the size of the time step used: smaller time step result in better time accuracy for transient problems and a more stable solutions, when explicit schemes are used.

The owner-neighbor addressing is applied when the sum term of equation (1.6) is evaluated. Should this term be evaluated naively *for each cell* using the outward directed normal S , the computation would be doubled for each cell face f once the loop reaches the adjacent cell (see Jasak 1996). However, the owner-neighbor addressing allows the FVM method in OpenFOAM to interpolate the face values only once, and then simply *apply the same contribution for both adjacent cells*:

$$\sum_f \phi_f U_f S = \sum_f^{owner} \phi_f U_f S_f - \sum_f^{neighbor} \phi_f U_f S_f, \quad (1.9)$$

where the sum is split into the *owner sum* and the *neighbor sum*. All the numerical calculations in OpenFOAM that involve face interpolation are based on looping over mesh faces. Cell values are accessed using owner-neighbor addressing of the cells, that has been described previously. This way the values ϕ_f and U_f in equation (1.9) are interpolated to the face centers once. Their contribution to the discretized term of two adjacent cells is also computed only once in a loop. It adds the same contribution to the face-owner cell, and deduct it from the face-neighbor cell. This way a significant amount of computational time is saved.



Face Interpolation

As mentioned in the previous sections, values of the discrete fields are stored in the CV centres. The discrete equation (1.6) makes use of the face values as well, when face values are interpolated from cell centred values. Evaluating face values is done by using *interpolation schemes*, which are one of the *main building blocks* of the OpenFOAM library. Interpolation schemes use values stored in cell centers \mathbf{C} to interpolate the values in the face centers \mathbf{C}_f . Using different interpolation schemes for the face centered value, ϕ_f will define the form of the discrete equation defined for each cell of the mesh. There are various interpolation schemes available, however explaining all of them is not within the scope of this book. To explain the basic working principle of an interpolation scheme, linear interpolation is chosen, also known as Central Differencing Scheme (CDS):

$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N, \quad (1.10)$$

Where f_x is the linear coefficient which is computed from the mesh geometry:

$$f_x = \frac{||\mathbf{fN}||}{||\mathbf{PN}||} \quad (1.11)$$

Equation (1.11) shows the role the *mesh geometry* plays in the final algebraic equation, assembled for a finite volume: large differences in cell size may lead to large errors in the face interpolation, that in turn reflect on the entire system of algebraic equations. A rigorous and detailed derivation of the interpolation errors of the arbitrary unstructured FVM is provided by Juretić 2004.

The positions of the points P , f and N are determined from the shape of the cells, as shown in figure 1.7. This plays a crucial role for the accuracy and stability of the numerical method. Equation (1.10) introduces the neighboring cell centre values into the algebraic equation of a cell.

To better illustrate how the algebraic equation is assembled for a cell, the example of a 2D finite volume is considered. With the labeled surrounding cells of the unstructured mesh shown in figure 1.6. For this volume, the discrete equation (1.6) takes on the following form:

$$a_9 \phi_0^n + a_4 \phi_4^n + a_3 \phi_3^n = 0 \quad (1.12)$$

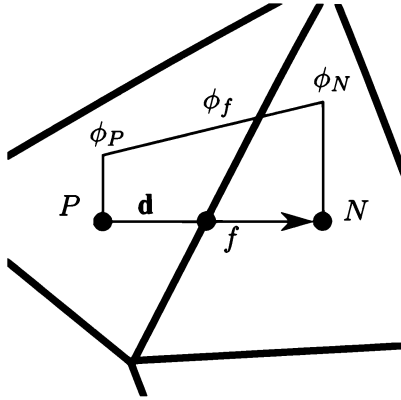


Figure 1.7: Schematic representation of the central differencing interpolation scheme applied on a 2D triangular mesh.

In this example, dependant variables are: ϕ_0 , ϕ_4 , and ϕ_3 for the cell 0 when an implicit temporal discretization scheme is applied. The number of the dependant variables in the algebraic equation is determined by the cell shape, since it determines the number of adjacent cells which take part in the assembly of the discrete advective term.

1.3.3 Boundary Conditions

The only thing missing from this brief description of the FVM is how boundary faces are treated. Such cell faces are highlighted by the thick line in figure 1.8 and in the OpenFOAM terminology, all boundary faces are called `boundaryField`. In that case, the variable cannot be made a dependant variable of the system, it needs to be prescribed. This is the reason why boundary conditions need to be prescribed, which can be further explained when observing the expanded discrete equation (1.8).

While expanding the sum term, the summation will come across a cell face which is the boundary face. If this face is marked with b , something like $\phi_b \mathbf{U}_b \mathbf{S}_b$ must be computed and added to the rest of the sum. Since there is only one cell next to a boundary face, this cell is - by definition - the owner of that face. The way of prescribing the boundary values are diverse, but the most basic ones are either `fixedValue` or `zeroGradient`. More details on this are provided in chapter 10. For the



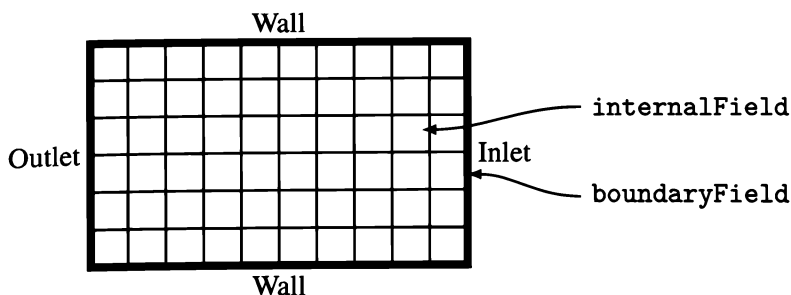


Figure 1.8: Example of a simple 2D channel flow with the inlet on the right side and the outlet on the left hand side. The other remaining two boundaries are assumed to be walls.

`fixedValue` boundary condition, the procedure is simple: prescribe the ϕ_b value as well as the boundary velocity \mathbf{U}_b . The simplest form of the boundary condition, is the so-called Neumann - or "natural" - boundary condition, which prescribes a zero gradient of the property at the domain boundary:

$$\nabla\phi(\mathbf{x}_b) = 0, \quad (1.13)$$

and this is the condition that is used to compute the value of the property at the boundary face b , using the Taylor series approximation:

$$\phi_P \approx \phi_b + \nabla\phi(\mathbf{x}_b)\delta x \approx \phi_b, \quad (1.14)$$

which results in the boundary value taking on the value from the cell. The boundary contribution to the algebraic equation for a zero-gradient boundary condition on the boundary face b will end up in the coefficient next to the cell value in the new time step: a_0 in equation (1.12). Various boundary conditions are implemented in OpenFOAM and all of them either prescribe the boundary value or the gradient, or a combination of both, on a set of boundary mesh faces.

WARNING

More information on how boundary conditions are implemented and actually used is given in chapter 8. The relation between the fields and the boundary conditions is part of this chapter as well.

1.3.4 Solving the System of Algebraic Equations

Equation (1.6) presents an example of how a PDE, like the equation (1.2), can be converted using equation discretization and interpolation schemes on the existing mesh into an algebraic equation. Implicit temporal discretization will result in a single algebraic equation being assembled per each cell. When the fully explicit temporal discretization is used, all values of the dependent variables in equation (1.12) apart from ϕ_0 are evaluated in the old timestep. As a result of that, the value in cell 0 in the old time step can be directly computed. Equations assembled for all CVs form a system of algebraic equation that needs to be solved for dependant variables, stored in the each cell and its respective neighbor cells.

A system of algebraic equations generated this way usually is very large: its size is directly proportional to the number of cells. One look at the fully expanded algebraic equation for example cell 1 shows how the cell indexing resulting from the mesh generation process determines the structure of the coefficient matrix of the algebraic system of equations. If a different ordering of cells is present, than the one shown in figure 1.3a, the coefficients and the dependant variables for the volume 1 would have different indexes.

In order to solve an algebraic system of equations, a square matrix of coefficients is required. No rows or columns may be defined as linear combinations of each other. This means that the true length of the fully expanded example equation (1.12) will be equal to the number of mesh cells. Which in turn results in a large number of equations for the entire mesh, that are assembled and solved in a matrix:

$$\underline{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b} \quad (1.15)$$

where $\underline{\mathbf{A}}$ is the coefficient matrix, \mathbf{x} represents unknowns of the system and \mathbf{b} denotes the source vector of the system.

Each row represents the connection of the particular cell to the other cells. As one cell does not possess a lot of direct connections to the remaining cells, only few columns in that row do actually have a value. The remaining columns are filled with zeros. Applied to the example,



this means that the rest of equation 1.12 is filled with zeros for all the cells of the mesh that have no direct relation to the example cell 1. This is why the final coefficient matrix $\underline{\mathbf{A}}$, assembled for an unstructured mesh is a *sparse* matrix: a matrix filled mostly with zeros.

After the system of equations is assembled, it has to be solved. In principle, the solving process itself can be either a *direct* or *iterative* method.

Direct Methods

A popular example for the *direct* methods is the Gauss elimination, that allows to obtain the solution of equation (1.15) in a direct manner, by rearranging the matrix. Unfortunately this number of those rearrangements is proportional to n^3 , with n being the size of the matrix (see Ferziger and Perić 2002). This renders this method unfeasible for large matrices, which usually occur in today's CFD simulations. Especially because the coefficient matrix is sparse and after applying the upper triangulation of the Gauss elimination, the matrix is not sparse anymore, which accounts for the slowness of the method.

Iterative Methods

These methods are essential for non-linear problems and opposed to the direct methods, although their accuracy is not as good. The solving process is started at a guessed solution and the final solution is achieved using an iterative approach. Ferziger and Perić (2002) formulate the system of equations for an iterative solving process like the following:

$$\underline{\mathbf{A}} \cdot \mathbf{x}^n = \mathbf{b} - \rho^n \quad (1.16)$$

The intermediate solution \mathbf{x}^n after n iterations does not suffice equation (1.15) and hence a residual ρ^n has to be introduced. It is always the aim to drive the residual towards zero. Since iterative solvers don't solve the system of equations in an absolutely accurate manner, the grade of accuracy must be defined somehow. This is where the residual comes into play, defining the difference between the exact solution and the current iteration.

Finding the most efficient iterative solver for the particular application is

necessary if one desires a fast convergence. OpenFOAM provides a large number of solvers, ranging from *preconditioned conjugate gradient* (PCG) to more sophisticated ones, such as *generalized geometric-algebraic multi-grid* (GAMG). To describe each of them in detail would be beyond the scope of this book and the reader is referred to Ferziger and Perić (2002) and Saad (2003) for details on this topic.

Improvement of Convergence

As real world applications tend to include various physical phenomenons and are usually of unsteady nature, one very common method to improve the convergence of the simulation is to employ *under-relaxation*.

Under-relaxation allows the user to define a blending factor α between the old and the new solution. An $\alpha = 1$ results with no effectively under-relaxation - the new solution is completely assembled with the result for this time step. Setting $\alpha = 0$ practically disables the development of the solution, as it is entirely composed by the previous solution, so choosing this value is of no merit. Values within the interval $(0, 1)$ define the blending between both solutions used to set the new solution of the system.

1.4 Overview of the Organization of the OpenFOAM Toolkit

WARNING

The OpenFOAM-2.2.x version that is used for the book, consisted of 7053 source files, totalling in approximately 1275306 source code lines. This excludes the configuration scripts, build system files as well as standard OpenFOAM headers placed in *. [CH] source files.

The OpenFOAM toolkit consists of many different libraries, stand-alone solvers, and utility programs. In order to establish some orientation around this massive and often intimidating code base, a look at the contents of the root OpenFOAM directory is taken in the following.

Contents of the OpenFOAM directory:



- applications** Source code for solvers, utilities, and auxiliary testing functions. Solver code is organized by function such as `/incompressible`, `/lagrangian`, or `/combustion`. Utilities are organized similarly into mesh, pre-processing, and post-processing categories among others.
- bin** Bash (not C++ binaries) scripts of with a broad array of functions: checking the installation (`foamInstallationTest`), executing a parallel run in debug mode, (`mpirunDebug`), generating an empty source code template (`foamNew`) or case (`foamNewCase`), etc.
- doc** User's Guide, the Programmer's Guide, and the Doxygen generation files.
- etc** Compilation and runtime selectable configuration controls flags for the entire library. Numerous installation settings are set in `/etc/bashrc` including which compiler to use, what MPI library to compile against, and where the installation will be placed (user local or system wide).
- platforms** Compiled binaries stored based on precision, debug flags, and processor architecture. Most installations will only have one or two sub-folders here which will be named according to the compilation type. For example, `linux64GccDP0pt` can be interpreted as follows:
 - `linux` operating system type
 - `64` processor architecture
 - `Gcc` compiler used (`Gcc`, `Icc`, `CLang`)
 - `DP` float precision (double precision (`DP`) vs. single precision (`SP`))
 - `0pt` Compiler optimization or debug flag. (Optimized (`Opt`) vs. Debug (`Debug`) vs. Profile (`Prof`))
- src** The bulk of the source code of the toolkit. Contains all of the CFD library sources including finite volume discretization, transport models, and the most basic primitive structures such as scalars, vectors, lists, etc. The main CFD solvers within the `applications` folder use the contents of these libraries to function.
- tutorials** Pre-configured cases for the various available solvers. The tutorials are useful for seeing how cases are set up for each solver. Some cases illustrate more complex pre-processing operations such as multi-region decomposition for solid-fluid heat transfer or Arbitrary Mesh Interface (AMI) setup.
- wmake** The bash based script, `wmake`, is a utility which configures and calls the C++ compiler. When compiling a solver or a library with

`wmake`, information from `Make/files` and `Make/options` is used to include headers and link other supporting libraries. A `Make` folder is required to use `wmake`, and thus to compile most OpenFOAM code.

The OpenFOAM library is described in depth from a software design perspective in chapter 5. Different paradigms of the C++ programming language and how they are used to make OpenFOAM such a modular and powerful CFD platform are explained there.

1.5 Summary

Using the OpenFOAM framework, as well as any other CFD framework, can be complicated because it usually entails having a solid understanding of physics, numerics, and engineering workflow. The open source nature of OpenFOAM brings a substantial amount of *free* power through flexibility to its user. Along with that power, a substantial complexity is involved for those CFD engineers that are used to working with a Graphical User Interface (GUI) and do not usually handle such an abundant choice of simulation parameters. As a first step towards the successful use of OpenFOAM, in this chapter the considerations when modeling a flow problem using numerical simulations are provided and main steps of the CFD analysis are described. This is followed by a brief overview of the FVM in OpenFOAM, so as to provide grounds for understanding elements of the OpenFOAM framework described throughout the rest of the book, such as: boundary conditions, discretization schemes, solver applications, etc. Finally, an organizational overview of the OpenFOAM framework is provided.

Further reading

Adaptive Mesh Refinement - Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3-5, 2003 (Lecture Notes in Computational Science and Engineering) (2005). 1st ed. Lecture notes in computational science and engineering, 41. Springer.

Ferziger, J. H. and M. Perić (2002). *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer.



- Jasak (1996). "Error Analysis and Estimatio for the Finite Volume Method with Applications to Fluid Flows". PhD thesis. Imperial College of Science.
- Jasak, Hrvoje, Aleksandar Jemcov, and Željko Tuković (2007). "Open-FOAM: A C++ Library for Complex Physics Simulations". In:
- Juretić, F. (2004). "Error Analysis in Finite Volume CFD". PhD thesis. Imperial College of Science.
- Rusche, Henrik (2002). "Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions". PhD thesis. Imperial college of Science, Technology and Medicine, London.
- Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems, Second Edition*. 2nd ed. Society for Industrial and Applied Mathematics. URL: http://www-users.cs.umn.edu/~saad/PS/all_pdf.zip.
- Ubbink, O. (1997). "Numerical prediction of two fluid system with sharp interfaces". PhD thesis. Imperial College of Science.
- Versteeg, H. K. and W. Malalasekera (1996). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method Approach*. Prentice Hall.
- Weller, H. G. et al. (1998). "A tensorial approach to computational continuum mechanics using object-oriented techniques". In: *Computers in Physics* 12.6, pp. 620–631.



2

Geometry Definition, Meshing and Mesh Conversion

Before going into the details of this chapter, some notions have to be put into context. A *geometry* in the CFD context is essentially a three dimensional representation of the flow region. A *mesh* on the other hand can have multiple meanings, but the three dimensional *volume mesh* is typically the one considered here. There are small variations to this, for example, a *surface mesh*, which is the discretization of a surface. As could be expected for surface discretization, planar areal elements are used as opposed to volume elements for the volume mesh. For complex geometries, proper definition of the surface mesh can be essential.

From a CFD perspective, it is the geometry relevant to each particular flow problem, that is of interest. Thinking of an aerodynamic simulation of the flow around a car, the interior of the car is generally of no interest, as it does not contribute to the overall flow in a significant manner. Therefore, only the details on the outside of the car's body are relevant and need to be resolved sufficiently in the spatial discretization.

This chapter outlines how to create a mesh from scratch, how to convert meshes between formats, and reviews various utilities for manipulating a

mesh after creation.

2.1 Geometry Definition

It is important to distinguish between the actual mesh geometry and the geometry that comes out of a Computational Aided Design (CAD) program. Though some words on the general mesh connectivity have been spent in the previous chapter, an overview of how the actual mesh is stored in the file system is given here. There are three main directories in a standard OpenFOAM case: `0`, `constant` and `system`. The first (`0`) stores the initial conditions for the fields and the last (`system`) stores settings related to the numerics and general execution of the simulation. For this chapter the `constant` directory is under consideration as it stores the mesh including all spatial and connectivity related data. Additional details on the OpenFOAM case structure are provided in chapter 3.

As long as static meshes are used, the computational grid is *always* stored in the `constant/polyMesh` directory. Static meshes, in this context, are meshes which are not changed over the course of the simulation either through point displacement or connectivity changes. Mesh data is naturally located here because it is assumed to be constant, hence the `constant` folder. From a programming perspective it is described as a `polyMesh` which is a general description of an OpenFOAM mesh with all its features and restrictions. For a given static mesh case the mesh data will be stored in `constant/polyMesh`. The typical mesh data files found here include: `points`, `faces`, `owner`, `neighbour` and `boundary`.

Of course, the contained data must be valid, to define a mesh properly. The `pitzDaily` tutorial of the `potentialFoam` solver serves as an example in the following discussion, which can be found by issuing the following commands:

```
?> tut
?> cd basic/potentialFoam/pitzDaily
```

After inspecting the content of the `polyMesh` directory, it is clear that it does not contain the required mesh data, yet. Solely the `blockMeshDict` is present in this tutorial, but executing `blockMesh` in the case directory generates the mesh and associated connectivity data:



```
?> ls constant/polyMesh
blockMeshDict boundary
?> blockMesh
?> ls constant/polyMesh
blockMeshDict boundary faces neighbour owner points
```

Users having worked with CFD codes before, especially with codes that are based upon structured meshes, may be missing the per-cell addressing. Rather than constructing the mesh on a per-cell basis, the unstructured FVM method in OpenFOAM constructs the mesh on a per-face basis. For more information on this, the reader is referred to the *owner-neighbour addressing* section of chapter 1.3. The following list illustrates the purposes of each file in `constant/polyMesh`.

points defines all points of the mesh in a `vectorField`, where their position in space is given in meters. These points are not the cell centres **C**, but rather the corners of the cells. To translate the mesh by 1 m in positive *x*-direction, each point must be changed accordingly. Altering any other structure in the `polyMesh` sub-directory for this purpose is not required, but this is covered by section 2.4.

A closer look at the `points` can be taken by opening the respective file with a text editor. To keep the output limited, the header is neglected and only the first few lines are shown:

```
?> head -25 constant/polyMesh/points | tail -7
25012 // Number of points
(
(-0.0206 0 -0.0005) // Point 0
(-0.01901716308 0 -0.0005) // Point 1
(-0.01749756573 0 -0.0005)
(-0.01603868134 0 -0.0005)
(-0.01463808421 0 -0.0005)
```

The `points` contain nothing more than a list of 25012 points. This list does not require to be ordered in any way although it can be. Additionally, all elements of the list are *unique*, meaning that point coordinates cannot occur multiple times. Accessing and addressing those points is performed via the list position in the `vectorField`, starting with 0. The position is stored as a `label`.

faces composes the faces from the points by their position in the `points` `vectorField` and stores them in a `labelListList`. This is a nested list, containing one element per face. Each of these elements is in turn a `labelList` of its own, storing the labels of the points

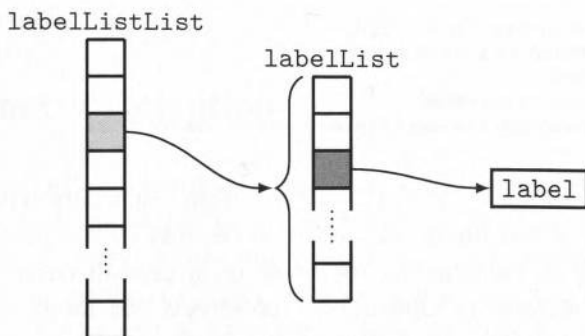


Figure 2.1: Graphical representation of a `labelListList`, used to represent the faces

used to construct the face. Figure 2.1 shows a visualization of the `labelListList` construct.

Each face must consist at least of three points and its size is followed by a list of point labels. On a face, every point is connected by a straight edge to its neighbours *OpenFOAM User Guide* 2013. Using the points which define the face, the surface area vector \mathbf{S}_f can be calculated where the direction is determined by the right-hand-rule.

Again, only the first few lines of the faces are shown to keep things short:

```
>> head -25 constant/polyMesh/faces | tail -7
49180 // Number of faces as labelListList
(
  4(1 20 172 153) // Face 0 with it's four point labels as labelList
  4(19 171 172 20)
  4(2 21 173 154)
  4(20 172 173 21)
  4(3 22 174 155)
  ...
)
```

As can be seen from the first line of the output, the mesh consists of 49180 faces of which only a subset is shown above. Similar to the face list's length of 49180, the length of each `labelList` is stated before the lists starts. Hence all faces shown here are built from 4 points, referred to by their position in the `points` list.

owner is again a `labelList` with the same dimension as the list storing the faces. Because the faces are already constructed and stored in

the faces list, their affiliation to the volume cells must be defined. Per definition, a face can only be shared between two adjacent cells. The *owner* list stores which face is owned by which cell, which in turn is decided based on the cell label. The cell with the lower cell label owns the face and the other remaining face is considered the neighbor. It instructs the code that the first face (index 0 in the list) is owned by the cell with the label that is stored at that position. A closer look at the *owner* file reveals that the first two faces are owned by cell 0 and the next two faces are owned by cell 1.

```
?> head -25 constant/polyMesh/owner | tail -7
49180
(
0
0
1
1
```

The ordering of this list is the result of the *owner-neighbour-addressing*, that was presented extensively in the previous chapter.

neighbour has to be regarded in conjunction with the *owner* list, as it does the opposite of the *owner* list. Rather than defining which cell owns each particular face, it stores its neighbouring cell. Comparing the *owner* with the *neighbour* file reveals a major difference: The *neighbour* list is significantly shorter. This is due to the fact that boundary faces don't have a neighbouring cell:

```
?> head -25 constant/polyMesh/neighbour | tail -8
24170
(
1
18
2
19
```

Again, the working principle of the *owner-neighbour-addressing* is explained in the previous chapter.

boundary contains all of the information concerning the mesh boundaries in a list of nested subdictionaries. Boundaries are often referred to as *patches* or *boundary patches*. Similar to the previous components of the mesh, only some relevant lines are shown:

```
?> head -25 constant/polyMesh/boundary | tail -8
5
(
  inlet // patch name
  {
```

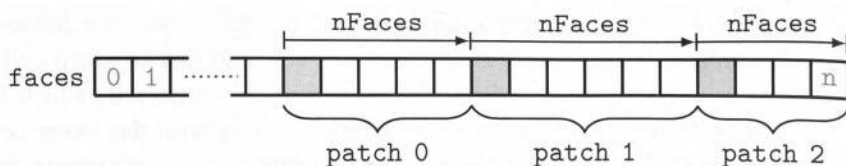


Figure 2.2: Working principle of the boundary addressing in OpenFOAM for a mesh with 3 patches and n faces. Grey elements denote the `startFace` of the particular patch.

```

    type      patch;
    nFaces    30;
    startFace  24170;
}

```

For the `pitzDaily` example used throughout this section, the boundary file contains a list of 5 patches. Each patch is represented by a dictionary, lead by the patch name. The information included in the dictionary includes: the patch type, number of faces and starting face. Due to the sorting of the faces list, faces belonging to a certain patch can be addressed quickly and easily using this convention.

The addressing method for the boundary faces is illustrated in Figure 2.2. By design, all faces which don't have a `neighbour` are collected at the end of the faces list where they are sorted according to their owner patch. All faces that are boundary faces must be covered by the boundary description.

From a user's perspective, neither `points` nor `faces`, `owner` and `neighbour` will need to be touched or manipulated manually. If they are changed manually, this will most certainly destroy the mesh. The boundary file, however, may need to be altered for certain setups depending on your workflow. The most likely reason for changing the boundary file is to alter a patch name or type. It may be considerably easier to make this change here rather than re-running the respective mesh generator.

Now that the basic structure of an OpenFOAM mesh is explained, the boundary patch types will be reviewed. There are several patch types that can be assigned to a boundary, some of which are more common than

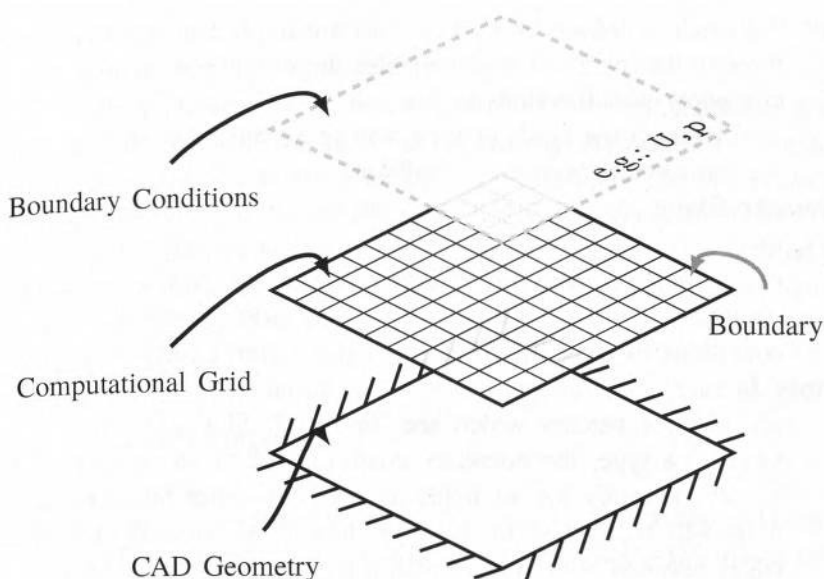


Figure 2.3: Illustration of the differences and relations between a CAD geometry, the computational grid and its boundaries, and the boundary conditions applied to those boundaries.

others. It is important to distinguish between the boundary (or patch) and the boundary *conditions* (see figure 2.3).

A patch is an outer boundary of the computational domain and it is specified in the boundary file, hence being a *topological* property. The logical connection between the boundaries and the CAD geometry is that the surface of both should be as identical as possible. Expressed in terms of mesh topology, it is a collection of faces that solely have an *owner* cell and no *neighbour* cell. In contrast to the patch, boundary conditions are applied to the patches for each of the fields (U, p , etc...) separately. The patch types are:

patch Most patches (boundaries) can be described by the type patch, as it is the most general description. Any boundary condition of Neumann, Dirichlet or Cauchy can be applied to boundary patches of this type.

- wall** If a patch is defined as wall, it does not imply that there is no flow through that patch. It solely enables the turbulence models to properly apply wall functions to that patch (see chapter 7). Preventing a flow through the patch of type wall must still be explicitly defined via the velocity boundary condition.
- symmetryPlane** Setting the patch type to `symmetryPlane` declares it to act as a symmetry plane. No other boundary conditions can be applied to it except `symmetryPlane`, and must be applied for *all* fields. For more information on the various available boundary conditions in OpenFOAM is given in chapter 10.
- empty** In case of a two-dimensional simulation, this type should be applied to the patches which are "in-plane". Similar to the `symmetryPlane` type, the boundary conditions of those patches have to be set to `empty` for all fields as well. No other boundary conditions will be applied to those patches. It is essential that all cell edges between both empty patches are parallel, otherwise an accurate two-dimensional simulation is not possible.
- cyclic** If a geometry consists of multiple components that are identical (e.g. a propeller blade or a turbine blade), only one needs to be discretized and treated as if it were located between identical components. For a four bladed propeller this would mean that only one blade is meshed (90° mesh) and by assigning a cyclic patch type to the patches with normals in tangential direction. These patches would then act as being physically coupled.
- wedge** This boundary type is similar to a cyclic patch, only specifically designed for cyclic patches which form a 5° wedge.

From an execution and compatibility stand-point, it does not matter how the `polyMesh` structure is created, so long as the mesh data is valid. While there are various mesh generation tools packaged with OpenFOAM, external third-party meshers can be used so long as a valid conversion or output can be made.

In addition to the above mentioned *essential* components of an OpenFOAM mesh, there are various optional mesh constructs which may only be used for particular applications.



Sets and Zones

Sets are essentially nothing more than a list of labels which address some elements of the mesh. The selection is usually performed by the tool `setSet` and based on boolean operations. Here, cells, points, or faces, can be selected and combined to a set. The set may be called `cellSet`, `faceSet` or `pointSet`, depending on what data type it is comprised of.

Zones on the other hand represent a subset of the mesh, that supports a lot of mesh related operations natively. Both groups are stored in the `constant/polyMesh` directory.

2.1.1 CAD Geometry

Importing a geometry that has been generated in an external CAD software is a regular task for any CFD engineer. In OpenFOAM this is commonly performed with `snappyHexMesh`, however, the usage of this mesh generator will be explained later on. The only important concept for now is that this section only deals with the import of Stereolithography (STL) files. Other file types are supported and work in a similar fashion. STL is a file format that can store the surfaces of geometries in a *triangulated* manner. Both binary and ASCII encoded files are possible, but for sake of simplicity we will work with ASCII encoding.

As an example of a STL file, the following snippet shows an STL surface comprised of only one triangle:

```
solid TRIANGLE
  facet normal -8.55322e-19 -0.950743 0.30998
    outer loop
      vertex -0.439394 1.29391e-18 -0.0625
      vertex -0.442762 0.00226415 -0.0555556
      vertex -0.442762 1.29694e-18 -0.0625
    endloop
  endfacet
endsolid TRIANGLE
```

In this example, only one solid is defined which is named `TRIANGLE`. A STL file may contain multiple solids which are defined one after the other. Each of the triangles that compose the surface has a normal vector and three points.

The drawback of using ASCII STL files is that their filesize tends to grow rapidly with increasing resolution of the surface. Edges are not included explicitly because only triangles are stored in the file. Because of this, identifying and extracting feature edges from an STL is sometimes a challenging task.

An advantage of using STL as a file format is that one obtains a triangulated *surface mesh*, which by definition will always have planar surface components (triangles).

2.2 Mesh Generation

There are multiple open source mesh generators designed specifically for OpenFOAM, spread between the two main development branches. This includes `blockMesh`, `snappyHexMesh`, `foamyHexMesh`, `foamyQuadMesh`, and `cfMesh`. There are a few remaining tools, `extrudeMesh` and `extrude2DMesh`, but are not addressed in this section as they are not used by most OpenFOAM users. `blockMesh` and `snappyHexMesh` will be addressed briefly in this section with a review of their usage and their working principles. In a general sense, the purpose of the mesh generators is to generate the `polyMesh` files described in the previous section in a user friendly fashion. Both mesh generators have similar input and output in that they read in a dictionary file and write the final mesh to `constant/polyMesh`.

2.2.1 `blockMesh`

When calling the executable `blockMesh` the `blockMeshDict` is read automatically from the `constant/polyMesh` directory, where it must be present.

`blockMesh` generates block-structured hexahedral meshes which are then converted into the arbitrary unstructured format of OpenFOAM. Generating grids with `blockMesh` for complex geometries is often a very tedious and difficult task and sometimes impossible. The effort that the user has to spend generating the `blockMeshDict` increases tremendously for complex geometries. Therefore, only simple meshes are typically generated using `blockMesh` and the discretization of the actual geometry is then



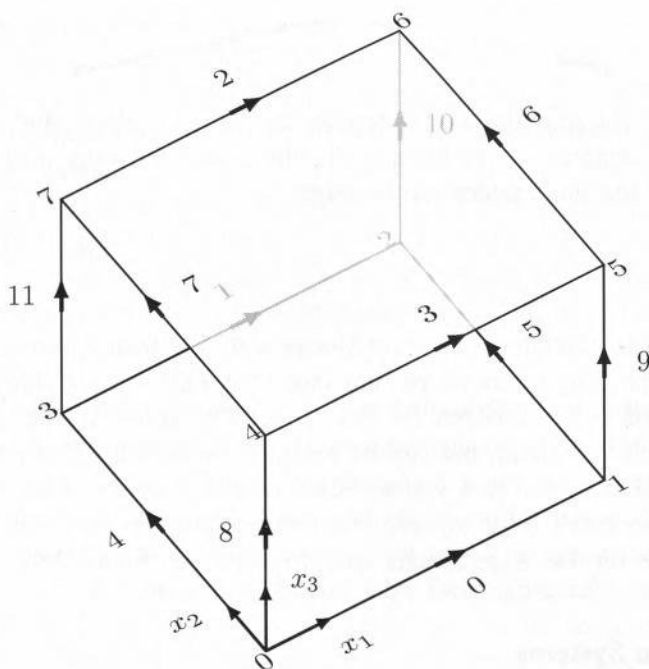


Figure 2.4: A blockMesh base block with vertex and edge naming convention.

shifted over to `snappyHexMesh`. This makes `blockMesh` a great tool for generating meshes which either consist of a fairly simple geometry, or to generate background meshes for `snappyHexMesh`.

An example block that `blockMesh` uses to construct the mesh is shown in figure 2.4. Each block consists of 8 corners called *vertices*. The hexahedral block is built from these corners. Edges, as indicated in figure 2.4, connect vertices with each other. Finally, the surface of the block is defined by patches though those have only to be specified explicitly for block boundaries that don't have a neighbouring block. Boundaries between two blocks must not be listed in the patch definition, as they are - by definition - not patches. The length and number of nodes on the particular edges must match in order to stay topologically consistent. Boundary conditions for the actual simulation will be applied later on those patches.



Figure 2.5: The gray dashed arc represents edge of a block, the black line denotes the resulting cell edges, and the gray dots indicate the three nodes on the edge.

Note that it is possible to generate blocks with less than 8 vertices and to have non-matching nodes on patches (see *OpenFOAM User Guide* 2013), however, this is not covered by this guide. The edges of the block are straight lines by default, but can be replaced by different line types such as an arc, a polyline, or a spline. Choosing e.g. an arc does affect the shape of the block edge topology, but the connection between the final mesh points on that edge remain straight lines (see figure 2.5).

Coordinate Systems

The final mesh is constructed in the global (right-handed) coordinate system, which is Cartesian and aligned with the major coordinate axes: x , y , and z . This leads to a problem when blocks must be aligned and positioned arbitrarily in space. To circumvent this issue, each block is assigned its own right-handed coordinate system, which by definition does not require the three axis to be orthogonal. The three axes are labelled x_1, x_2, x_3 (see *OpenFOAM User Guide* 2013 and figure 2.4). Defining the local coordinate system is performed based on the notation shown in figure 2.4: Vertex 0 defines the origin, the vector between vertices 0 and 1 represents x_1 , x_2 and x_3 are composed of the vectors between vertices 0 and 2 and 0 and 4, respectively.

Node Distribution

During the meshing process, each block is subdivided into cells. The cells are defined by the nodes on the edges in each of the three coordinate axes of the block's coordinate system and follow the relationship given by:



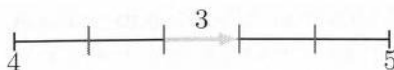


Figure 2.6: Illustration of the expansion ratio, with edge 3 used as an example

$$n_{\text{cells}} = n_{\text{nodes}} + 1 \quad (2.1)$$

It is up to the user to define in the `blockMeshDict` how many cells will appear on a certain edge. Cells on an edge can either be distributed uniformly or on a non-uniform spread based on a grading. There exist two types of gradings: **simpleGrading** and **edgeGrading** based. A **simpleGrading** describes the grading on an edge based on the size ratio of the last to the first cell on that particular edge (see figure 2.6):

$$e_r = \frac{\delta_e}{\delta_s} \quad (2.2)$$

If $e_r = 1$ all nodes are spaced uniformly on that particular edge, no grading is present. With an expansion ratio $e_r > 1$, the node spacing increases from start to end of the edge. From the C++ sources of `blockMesh` it can be found that the expansion ratio that is defined by the user (e_r) is scaled by the following relation:

$$r = e_r^{\frac{1}{1-n}} \quad (2.3)$$

Where n represents the number of nodes on that particular edge. By inserting equation (2.3) into equation (2.4), the relative position of the i -th node on an edge can be calculated.

$$\lambda(r, i) = \frac{1 - r^i}{1 - r^n} \quad \text{with } \lambda \in [0, 1] \quad (2.4)$$

Even though this might look too laborious to perform for all of the blocks in a `blockMeshDict`, this comes in handy when a smooth transition in the cell sizes between two adjoining blocks is required. In many cases, simple try and error usually suffices.

Defining the dictionary for a minimal example

As a small example on how the `blockMeshDict` is set up correctly, a cube of 1 m^3 in volume is discretized. A prepared case can be found in `chapter2/blockMesh` directory of the example case repository. The dictionary itself consists of one keyword and four sub-dictionaries. The first keyword is `convertToMeters` which is usually 1. All point locations are scaled by this factor, which comes in handy if the geometry is very large or very small. In any of those cases we would end up typing a lot of leading or trailing zeroes, which is a tedious task. By setting `convertToMeters` accordingly, we can save some typing. The first relevant line of the `blockMeshDict` is shown in listing 1.

Listing 1 Scaling factor in blockMesh

```
convertToMeters    1;
```

Secondly the vertices must be defined. It is important to remember that vertices in `blockMesh` are different from the points of the created `polyMesh`, though their definition is fairly similar. For the unit cube example the vertices definition is shown in listing 2.

Listing 2 Vertice setup for a unit cube

```
vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 1)
    (1 0 1)
    (1 1 1)
    (0 1 1)
);
```

From having a glance at the above definition, it is clear that the syntax is a list and similar to the list of points in the `polyMesh` definition. This



is due to the round brackets that indicate a list in OpenFOAM, whereas curly brackets would define a dictionary. The first four lines define all four vertices in the $x_3 = 0$ plane and the following do the same for the $x_3 = 1$ plane. Similar to the points in polyMesh, each element is accessed by its position in the list and not by the coordinates. Please note, that each vertex must be unique and hence only occur once in the list.

As a next step, the blocks must be defined. Figure 2.4 can be used as a reference. An example block definition for the unit cube is presented in listing 3.

Listing 3 Block setup for a unit cube

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (10 10 10) simpleGrading (1 1 1);
);
```

Again this is a list which contains blocks and is not a dictionary, due to the round brackets. The definition might appear odd at first glance, but is actually quite straight forward. The first word `hex` and the first set of round brackets containing eight numbers tells `blockMesh` to generate a hexahedron out of the vertices 0 to 7. These vertices are exactly those specified in the `vertices` section above and are accessed by their labels. Their order is not arbitrary, but defined by the *local block* coordinate system as follows:

1. For the local $x_3 = 0$ plane list all four vertex labels starting at the origin and moving according to the right-handed coordinate system.
2. Do the same for the local $x_3 \neq 0$ plane

It is possible to obtain a valid block definition by messing up the order of the vertex list in the particular block definition. The resulting block will either look twisted or uses incorrect global coordinate orientation. As soon as `blockMesh` and `checkMesh` are executed and the mesh is analyzed in a post-processor (e.g. `paraView`), this is detected.

The second set of round brackets defines how many cells are distributed in each particular direction of the block. In this case, the block possesses 10 cells for each direction. Changing the cell count to 2 cells in x_1 , 20

TIP

`checkMesh` is a native OpenFOAM tool to check the mesh integrity and quality, based on various criteria. If the output of `checkMesh` states that the mesh is not ok, it *must* get improved.

cells in x_2 and 1337 cells in x_3 , the block definition would look like this:

```
hex (0 1 2 3 4 5 6 7) (2 20 1337) simpleGrading (1 1 1);
```

The last remaining bit is the `simpleGrading` part in conjunction with the last set of numbers in the round brackets. This is the easiest way of defining a grading (or expansion ratio) as described before. The keyword `simpleGrading` in this case, defines the grading for all four edges in each of the three local coordinate system's axis directions to be identical. Hence each of the three numbers stated in the brackets after `simpleGrading` defines the grading for x_1 , x_2 and x_3 direction, respectively. Sometimes this is not versatile enough, though. This is where `edgeGrading` can be used. This more advanced grading approach is essentially the same as `simpleGrading`, but the grading for each of the 12 edges on a hexahedron can be specified explicitly. Therefore the last set of brackets would not list 3 numbers, but 3 times 4. Now, each edge can be set individually.

Saving the `blockMeshDict` and executing `blockMesh` afterwards, results in a valid mesh that looks similar to what is defined in the `blockMeshDict`. But `blockMesh` warns about undefined patches that are all put into the *defaultFaces* patch by default.

Specifying patches manually is done by defining them inside the list called `patches` and for the example patch 0, as shown in listing 4.

WARNING

There is a significant difference between various OpenFOAM versions, in terms of how the `blockMeshDict` is defined. As this book is based on the version 2.2 of OpenFOAM, the older versions are not considered.



Listing 4 Patch setup for an individual side.

```

patches
(
  XMIN
  {
    type patch;
    faces
    (
      (4 7 3 0)
    );
  }
);

```

This instructs `blockMesh` to generate a patch of type `patch` named `XMIN`, based on the face that is constructed from the vertices 4, 7, 3 and 0. Internally, the patch name is defined as word and this data type shows up in error messages regularly. How the vertices are ordered is not arbitrary, though. They need to be specified in a clockwise orientation, looking from *inside* the block. An image of the unit cube of our minimal example, consisting of 1000 small cubes is shown in figure 2.7, with highlighted `XMIN`, `YMIN` and `ZMAX` patches. The files used to generate this mesh can be found in the example repository under `chapter2/blockMesh`.

As stated earlier, the edges of a block are lines by default and thus the list containing the edge definitions is optional. Quite similarly, to the above defined blocks and patches, connecting two vertices by an arc instead of the default line would look like this:

```

edges
(
  arc 0 1 (0.5 -0.5 0)
);

```

Each item of the list containing the edge definitions starts with a keyword which indicates the type of edge, followed by the labels of the start and end vertex. In this example, the line is closed by the third point that is required to construct an arc. For any other edge shape (e.g. *polyLine* or *spline*), this point would be replaced by a list of supporting points.

An example of how inserting the above listed code alters the shape of

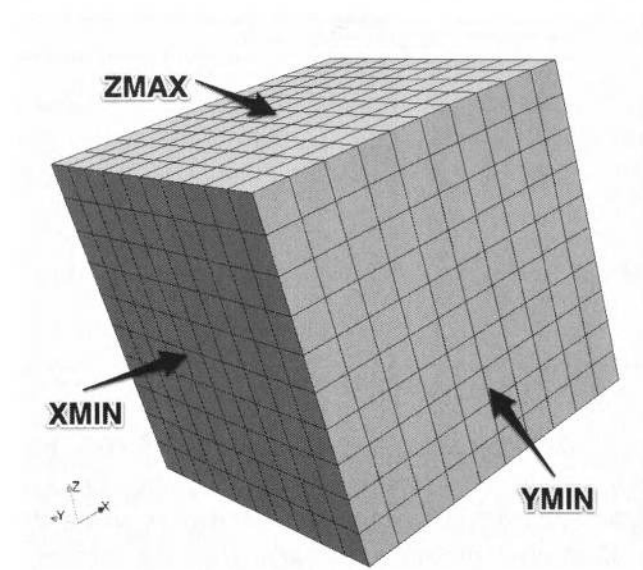


Figure 2.7: Unit cube meshed with `blockMesh` and an edge resolution of 10 cells in each direction

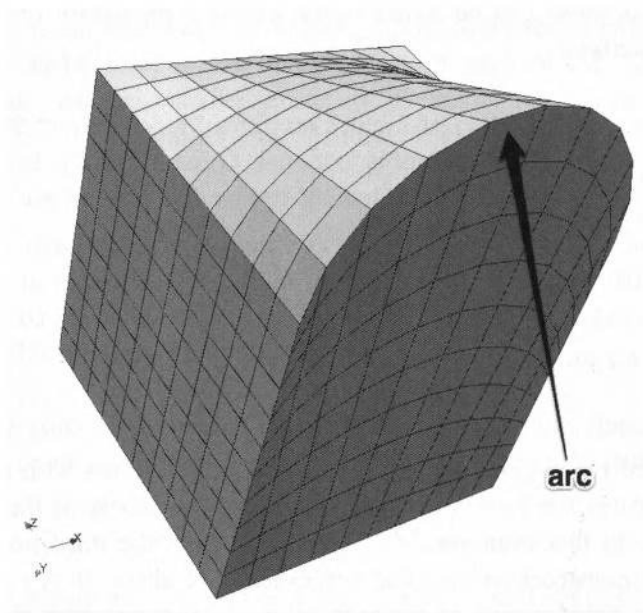


Figure 2.8: Unit cube meshed with `blockMesh` and an arc as one edge



the unit cube (see figure 2.7) is presented in figure 2.8.

To proceed with the `snappyHexMesh` section, you need to generate a unit cube consisting of 50 cells in each direction.

2.2.2 `snappyHexMesh`

Compared to `blockMesh`, `snappyHexMesh` may not require as much tedious work, such as adding and connecting blocks. On the other hand, one has less control over the final mesh. With `snappyHexMesh`, hexa-dominant meshes can be generated easily, needing only two things: a hexahedral background mesh and secondly one or more geometries in a compatible surface format. `SnappyHexMesh` supports local mesh refinements defined by various volumetric shapes (see table 2.1), application of boundary layer cells (prisms and polyhedras) and parallel execution.

`SnappyHexMesh` is a complex program and is controlled by a multitude of controlling parameters. Describing all of them in detail is beyond the scope of this book. Please read the *OpenFOAM User Guide 2013* in conjunction with this book, for a more thorough discussion of `snappyHexMesh`. Additional information can be found here: Villiers 2013.

The execution flow of `snappyHexMesh` can be split into three major steps which are executed successively. Each of these steps can be disabled by setting the respective keywords to `false` at the beginning of the `snappyHexMeshDict`. These three steps can be summarized as follows:

castellatedMesh This is the first stage and performs two main operations. First, it adds the geometry to the grid and removes the cells which are not inside the flow domain. Second, the existing cells are split and refined according to the user's specifications. The result is a mesh which consists only of hexahedrons that more or less resembles the geometry. However, the majority of mesh points which are supposed to be placed on the geometry's surface are not aligned with it. A screenshot of a later example at this stage of the meshing process is shown in figure 2.9.

snap By performing the snapping step, the mesh points in the vicinity of the surface are moved onto the surface. This can be seen in figure 2.10. During this process, the topology of those cells may get

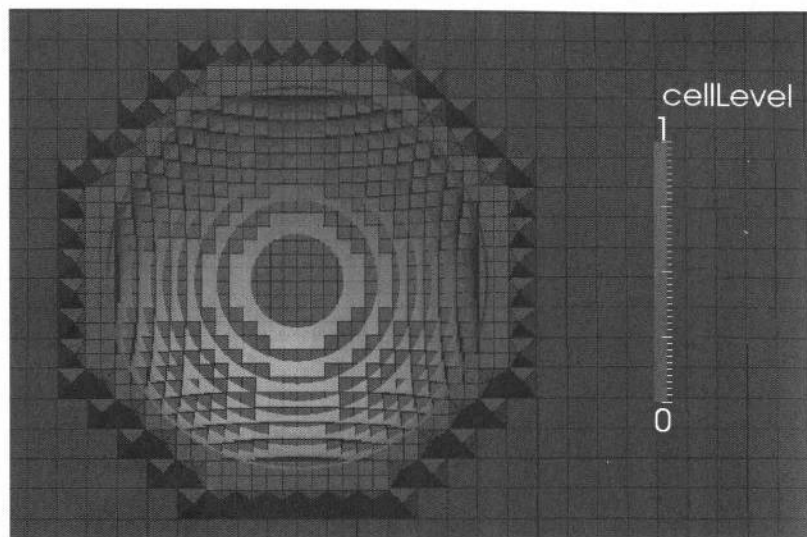


Figure 2.9: A STL sphere ($D = 0.25$ m) meshed with `snappyHexMesh` after the first meshing step. The hexahedrons are not aligned with the body surface, yet.

changed from hexahedrons to polyhedrons. Cells near the surface may be deleted or merged together.

addLayers Finally, additional cells are introduced on the geometry surface, that are usually used to refine the near wall flow (see figure 2.11). The pre-existing cells are moved away from the geometry in order to create space for the additional cells. Those cells are likely to be prisms.

All of the above settings and many more are defined in `system/snappyHexMeshDict`, the dictionary which contains all of the parameters required by `snappyHexMesh`. Several helpful tutorials can be found in the OpenFOAM tutorials directory under `meshing/snappyHexMesh`. Compared to other OpenFOAM dictionaries, the `snappyHexMeshDict` is very long and consists of many hierarchy levels which are represented by nested subdictionaries. One time step is written to the case directory, for each of the above mentioned steps (assuming you have a standard configuration). Each of the three steps will be addressed individually in the following section.

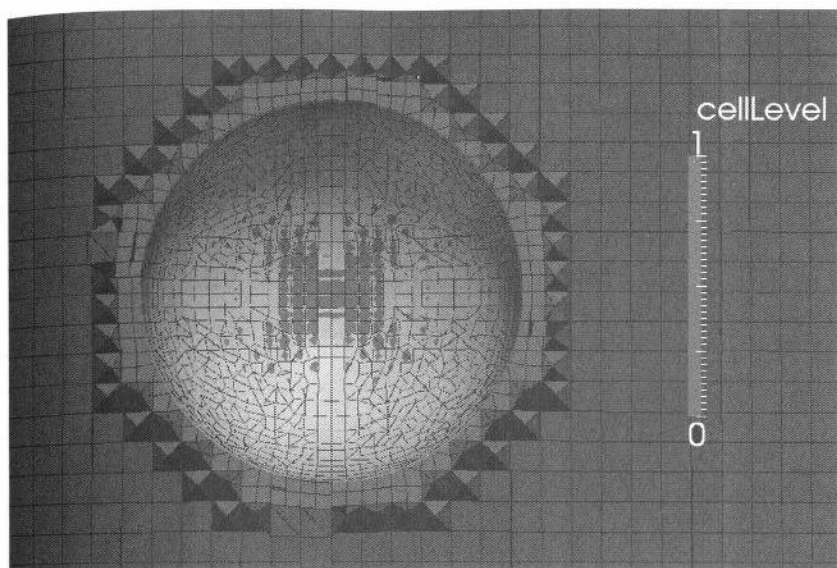


Figure 2.10: The same sphere as above but after the snapping process. All points are aligned with the body surface.

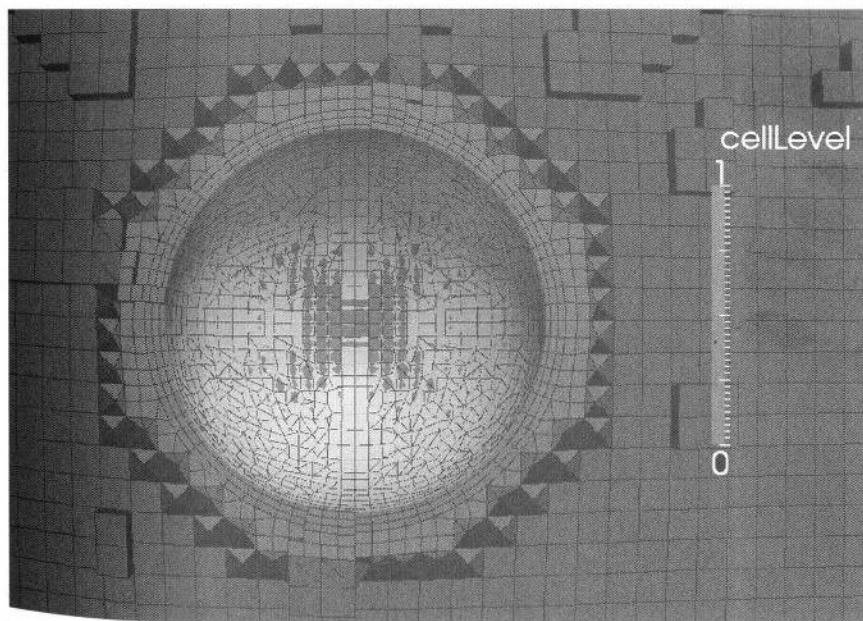


Figure 2.11: Prism layers are applied to the sphere surface, by extruding the surface.

Cell levels

Cell levels are used to describe the refinement status of a background mesh cell. When `snappyHexMesh` is started, the background mesh is read and all cells are assigned cell level 0 (blue cells in figure 2.11). If a cell is refined by one level, each of the edges are halved, generating eight cells from the previous one 'parent' cell. This method of refining is based on octrees and is only applicable for hexahedrons which is why a hexahedral background mesh is required by `snappyHexMesh`. With `snappyHexMesh` it is impossible to refine cells in only one direction, as this cannot be covered by octrees. Therefore they are refined - by definition - uniformly in all three spatial directions.

Defining the geometry

Before starting the meshing process, the geometry must be defined in the *geometry* subdictionary in the `snappyHexMeshDict`. Without the need to define anything in the `snappyHexMeshDict`, the existing mesh in `constant/polyMesh` is read automatically and serves as background mesh. If there is no such mesh available, or if it is not purely based on hexahedrons, `snappyHexMesh` will not run. For external flow simulations, the outer boundaries defined by the background mesh are not as important as for internal flows. They can hence be kept as defined by the background mesh, without spending further work on them. For internal flow simulations on the other hand, the outer shape of the background mesh is of no interest as it is defined by the actual geometry.

TIP

STL geometries can be generated using almost any CAD program. Paraview may be used to generate an STL representation of basic shapes such as cylinders, spheres, or cones. Under the *sources* menu, various shapes are available which can be exported using the *save data* entry, under *file* menu.

As a simple example, the unit cube mesh which was prepared in the previous section is reused and a sphere is inserted into it. The sphere is generated using a STL file, instead of the shapes listed in table 2.1. Loading a STL geometry can be done in a straight forward manner, by



Shape	Name	Parameters
Box	<code>searchableBox</code>	min, max
Cylinder	<code>searchableCylinder</code>	point1, point2, radius
Plane	<code>searchablePlane</code>	point, normal
Plate	<code>searchablePlate</code>	origin, span
Sphere	<code>searchableSphere</code>	centre, radius
Collection	<code>searchableSurfaceCollection</code>	geometries

Table 2.1: List of cell selection shapes

simply copying the geometry to `constant/triSurface` of the case and adding the following geometry subdictionary in the `snappyHexMeshDict`. An example of this is presented in listing 5.

Listing 5 Geometry definition in `snappyHexMesh`

```

geometry
{
    sphere.stl // Name of the STL file
    {
        type    triSurfaceMesh; // Type that deals with STL import
        name    SPHERE; // Name access the geometry from now on
    }
}

```

The lines above tell `snappyHexMesh` to read `sphere.stl` from `constant/triSurface` as a `triSurfaceMesh` and refer to the geometry contained in that STL as `SPHERE`. Some simple geometry objects can be constructed without the need to open any CAD program, right inside `snappyHexMesh`. A list of these geometrical shapes is compiled in table 2.1.

Any of the shapes listed in table 2.1 can be constructed in the `geometry` subdictionary by simply appending to the existing subdictionary. As an example, a box is to be added to the `geometry` subdictionary, which is constructed from a minimum and maximum point (see listing 6). When using this approach not that it is impossible to rotate the box straight away and it will always be aligned with the coordinate axis.

Similar to the STL definition, the leading string of the subdictionary

Listing 6 Definition of a searchableBox

```
smallerBox
{
    type    searchableBox;
    min     (0.2 0.2 0.2);
    max     (0.8 0.8 0.8);
}
```

that defines the `searchableBox` is the name that is used to access that geometry later on. Sometimes it is desirable to compose a geometry out of the shapes listed in Table 2.1, but treat it as one single geometry rather than multiple. This is where the `searchableSurfaceCollection` can be used. By using this approach on geometry components that already exist, surfaces can be combined, rotated, translated and scaled. In any case, combining SPHERE and `smallerBox` into one and scaling the `fancybox` up by a factor of 2 would look as listed in listing 7.

Listing 7 Complete example of the geometry subdictionary

```
geometry
{
    ...
    fancyBox
    {
        type searchableSurfaceCollection;
        mergeSubRegions true;
        SPHERE2
        {
            surface SPHERE
            scale  (1 1 1);
        }
        smallerBox2
        {
            surface smallerBox;
            scale  (2 2 2);
        }
    }
}
```

Setting up the castellatedMesh step

This is the first out of three steps during the execution of `snappy-HexMesh`. It includes the following two main steps: splitting the cells according to the user specifications and deleting cells that are outside the meshed region. A schematic of this process is given in figure 2.12.



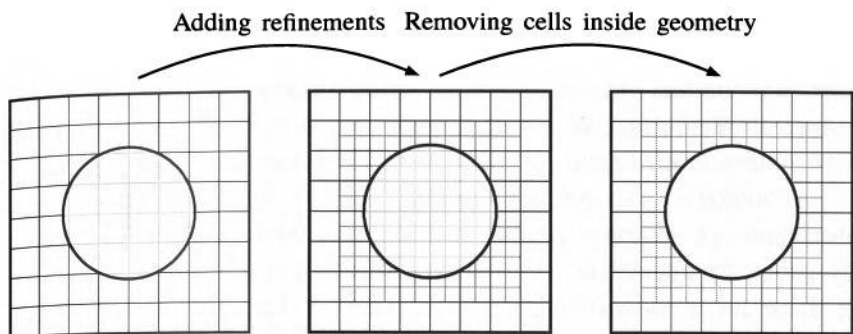


Figure 2.12: Schematic of the actions performed during the `castellatedMesh` step.

The existing background mesh (black in figure 2.12) is read from `constant/polyMesh`. Based on the parameters in the `castellatedMeshControls` subdictionary of the `snappyHexMeshDict`, the mesh is refined. It is important to distinguish between refinements that are defined by *geometry surfaces* and *volumetric refinement*. The *surface refinement* ensures that the boundary faces that represent the geometry are refined up to the defined level. It is important to note that this does not only affect the cells owning the particular, but also cells adjoining these cells. Therefore, surface refinement may appear somewhat similar to a volumetric refinement, however, it is distinctly different. Applying such a surface refinement to the SPHERE is controlled by entries in the `castellatedMeshControls`, shown in listing 8.

Listing 8 Example of surface refinements

```
castellatedMeshControls
{
    ...
    refinementSurfaces
    {
        SPHERE // Name of the surface
        {
            level (1 1); // Min and max refinement level
        }
    }
    ...
}
```

This refines the surface of the SPHERE to level 1. The two numbers between the round brackets define a minimum and maximum level of refinement for this surface. `snappyHexMesh` chooses between both, depending on the surface curvature: highly curved surface areas are refined to the higher level, lesser curved ones to the lower level.

Refinements in `snappyHexMesh` are not limited to surface definitions. Any geometry defined in the geometry subdictionary can serve as defining shape for a *volumetric refinement*. These volumetric refinements are called `refinementRegions` and are defined in a subdictionary with the same name, in the `castellatedMesh` controls. Compared to `refinementSurfaces`, `refinementRegions` offer a higher grade of versatility and thus require more options to be defined.

The mode has three options: `inside`, `outside` and `distance`. As the names suggests, `inside` only affects cells inside the selected geometry whereas `outside` does exactly the opposite. The third option, `distance`, is a combination of both and is calculated in outward *and* inward normal direction of the surface. In addition to modes, there is a `levels` option, which is more complex than for `refinementSurfaces`. It can already be guessed from the name, it does support an arbitrary number of levels. Each level must be defined in conjunction with a distance. With increasing position in the list, the levels must decrease and the distances must increase. Refining anything inside the `smallerBox` to level 1 can be done by adding the lines presented in listing 9.

Listing 9 Example of applying `refinementRegions`

```
castellatedMeshControls
{
    ...
    refinementRegions
    {
        smallerBox // Geometry name
        {
            mode    inside; // inside, outside, distance
            levels  ((1E15 1)); // distance and level
        }
    }
    ...
}
```

The above code uses a distance of 1×10^{15} m, in order to safely select

all cells inside the geometry.

Without specifying a point located inside the volume of the final mesh, it is impossible for `snappyHexMesh` to decide which side of the sphere the user wants to discretize. That is why the `locationInMesh` keyword must be defined in the `castellatedMeshControls` subdictionary, as well. This point must not be placed on a face of the background mesh. For our unit cube example, this point is defined in listing 10.

Listing 10 Definition of `locationInMesh`

```
locationInMesh (0.987654 0.987654 0.987654);
```

The next step is to adjust the parameters of the `snap` subdictionary in the `snappyHexMeshDict`.

Setting up the snapping step

Compared to the other two steps of `snappyHexMesh`, this does not require extensive user input. This step is responsible for aligning the purely hexahedral mesh faces with the geometry by introducing new points into the mesh and displacing them (see figure 2.10). This is a highly iterative process, which is the reason why there is not much user interaction required. The `snapControls` subdictionary of the `snappyHexMeshDict` is shown in listing 11.

Listing 11 Entries of `snapControls`

```
snapControls
{
    nSmoothPatch 3;
    tolerance 2.0;
    nSolveIter 30;
    nRelaxIter 5;

    // Feature snapping
    nFeatureSnapIter 10;
    implicitFeatureSnap false;
    explicitFeatureSnap true;
    multiRegionFeatureSnap false;
}
```

There are only iteration counters, tolerances and flags defined. Half of the parameters deal with snapping to the edges of the geometry, which

is not part of this description. A description of this can be found on Höpken and Maric 2013, however. Depending on the specifics of the case, increasing the iteration counters usually leads to a higher quality mesh, but also increases the meshing time significantly.

TIP

All of the parameters are explained in further detail in the `snappy-HexMeshDicts`, provided with OpenFOAM.

Setting up the `addLayers` step

All settings for the `addLayers` step are defined in the `addLayersControls` subdictionary of the `snappyHexMeshDict` (see listing 12). Any surface can be used to extrude prism layers from, regardless of its type. Firstly, the number of cell layers to be extruded per boundary needs to be specified via the `layers` subdictionary.

Listing 12 Example entries of `addLayersControls`

```
addLayersControls
{
    ...
    layers
    {
        "SPHERE_.*" // Patch name with regular expressions
        {
            nSurfaceLayers 3; // Number of cell layers
        }
    }
    ...
}
```

Each patch name is followed by a subdictionary that contains the `nSurfaceLayers` keyword. This keyword defines the number of cell layers that get extruded and is thus followed by an integer. In the above example, regular expressions are employed to match any patch names which start with `SPHERE_`. In this case it is only the sphere itself however the use of wildcard characters in this manner can greatly reduce setup time. A cross-section of the final mesh is shown in figure 2.11.

Various parameters of `snappyHexMesh`, especially those related to the layer extrusion, require adjustment in order to obtain a mesh that meets



your requirements. A few of those are explained briefly in the following.

relativeSizes can switch from absolute to relative dimensioning for the following values. By default it is `true`.

expansionRatio defines the expansion factor from one cell layer to the next one.

finalLayerThickness is the thickness of the last cell layer (furthest away from the wall), with respect to the next cell of the mesh or in absolute meters, depending on your choice for the **relativeSizes** parameter.

minThickness if a layer cannot be thicker than **minThickness**, it is not extruded.

In the example, the settings shown in listing 13 were employed.

Listing 13 Entries of `addLayersControls` used in the example

```
relativeSizes      true;
expansionRatio     1.0;
finalLayerThickness 0.5;
minThickness       0.25;
```

Finally `snappyHexMesh` must be executed in the case directory to begin the meshing process. Each step generates a new time step directory which contains the mesh at that particular stage. If you choose to make changes to your mesh by adjusting the parameters in the `snappyHexMeshDict`, remember to delete the old time steps before rerunning `snappyHexMesh`.

TIP

Another high quality mesh generator for OpenFOAM is *enGrid* and can be obtained freely from <http://engits.eu/en/engrid>.

2.2.3 cfMesh

The `cfMesh` library is a cross-platform library for automatic mesh generation that is built on top of OpenFOAM. It is compatible with all recent versions of OpenFOAM and foam-extend and is licensed under General

TIP

The application *cfMesh* is a distributed memory parallel OpenFOAM meshing tool which, like *snappyHexMesh*, takes in STL surfaces as an input. This package is developed by Creative Fields Ltd. and can be downloaded along with documentation at www.c-fields.com.

Public License (GPL). The library was developed by Dr. Franjo Juretić and it is distributed by *Creative Fields*, Ltd and is available for download from <http://www.c-fields.com>.

This section covers a brief overview of the library, options that govern the mesh generation process, as well as some of the utilities distributed with *cfMesh*. It is in no way a complete documentation of the project. More information is available on the aforementioned project web page.

The *cfMesh* library supports various 3D and 2D workflows, built by using components from the main library, which are extensible and can be combined into various meshing workflows. The core library is based on the concept of mesh modifiers, which allows for efficient parallelisation using both Symmetric Multiprocessor (SMP) and Distributed Memory Parallel (DMP) using message passing interface (MPI). In addition, special care has been taken on memory usage, which is kept low by implementing data containers (lists, graphs, etc.) that do not require many dynamic memory allocation operations during the meshing process.

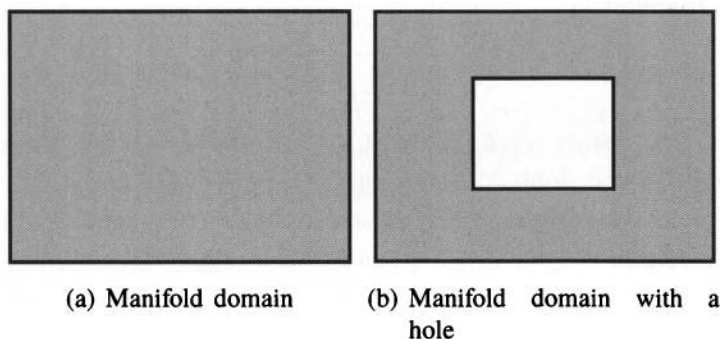


Figure 2.13: Allowed types of geometries in *cfMesh*

The meshing process in *cfMesh* is automatic, and it requires an input triangulation and dictionary with the various meshing parameters (settings). Once the surface mesh and the settings are given, the meshing process is started from the console, and it runs automatically without any user intervention. The library is optimised such that the meshing workflows require a small number of settings, and they have a simple syntax. Currently, *cfMesh* can create volume meshes inside a manifold, see figure 2.13, which does not need to be watertight.

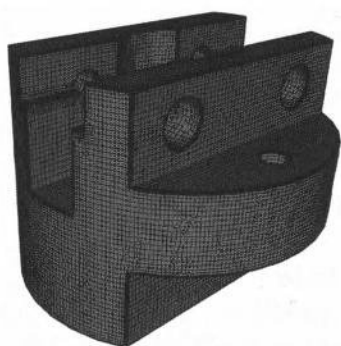
Available meshing workflows

All workflows are parallelised for shared memory machines and use all available CPU cores while running. The number of utilised cores can be controlled by the `OMP_NUM_THREADS` environment variable which can be set to the desired number of cores.

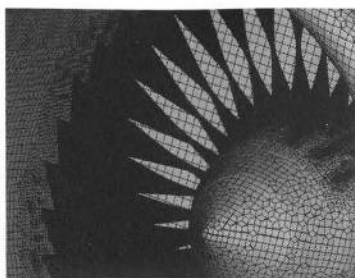
The available meshing workflows start the meshing process by creating a so-called mesh template from the input geometry and the user-specified settings. The template is later on adjusted to match the input geometry. The process of fitting the template to the input geometry is designed to be tolerant to poor quality input data which does not need to be watertight. The available workflows differ by the type of cells generated in the template.

Cartesian workflow generates 3D meshes consisting of predominantly hexahedral cells with polyhedra in the transition regions between the cells of different size. It is started by typing `cartesianMesh` in a shell window. By default, it generates one boundary layer which can be further refined on user request. In addition, this workflow can be run using MPI parallelisation, which is intended for generation of large meshes which do not fit into the memory of a single available computer.

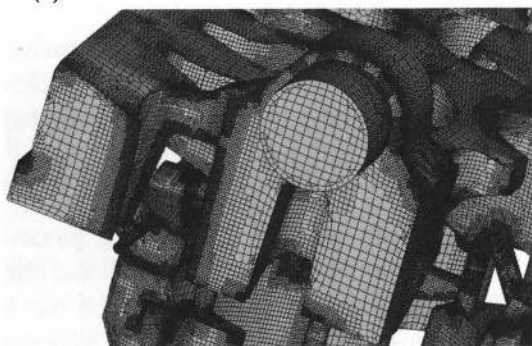
The workflow generates 2D cartesian meshes. The mesh generator is started by typing `cartesian2DMesh` in the console. By default, it generates one boundary layer, which can be further refined. This meshing workflow requires the geometry in a form of a ribbon, as shown in figure 2.15a, which span in the x-y plane and is extruded in the z direction.



(a) Socket



(b) Engine of an aeroplane



(c) Cooling jacket

Figure 2.14: 3D cartesian meshing

Tetrahedral workflow generates meshes consisting of tetrahedral cells, figure 2.16, and is started by typing `tetMesh` in a console. By default, it does not generate any boundary layers, and they can be added and refined on user request.

Input geometry

Geoemtries used by `cfMesh` are required to be defined in the form of a surface triangulation. For 2D cases, the geometry is given in a form of a ribbon of triangles with boundary edges in the x-y plane (other orientations are not supported). The geometry consists of the following entities:

List of points contains all points in the surface triangulation.



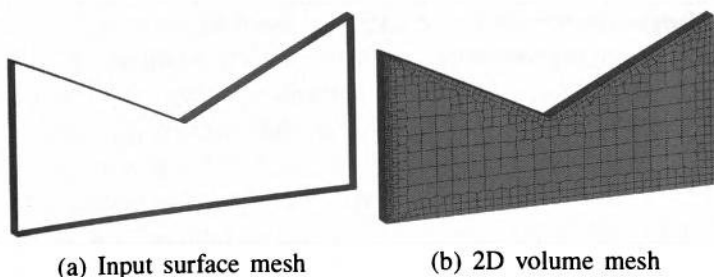


Figure 2.15: 2D cartesian meshing

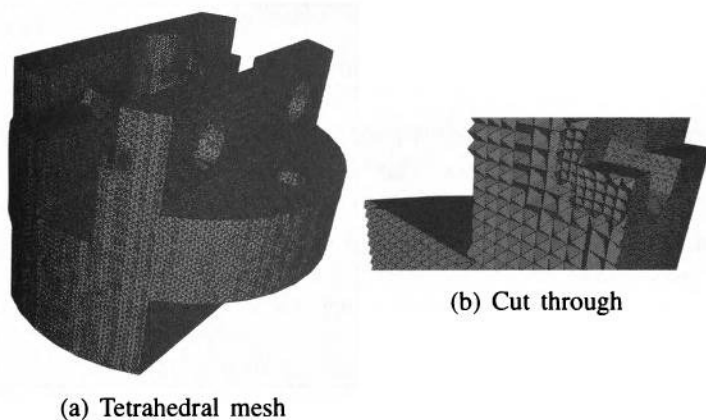


Figure 2.16: Tetrahedral meshing

List of triangles contains all triangles in the surface mesh.

Patches are entities which are transferred onto the volume mesh in the meshing process. Every triangle in the surface is assigned to a single patch, and cannot be assigned to more than one patch. Each patch is identified by its name and type. By default, all patch names and types are transferred to the volume mesh, and are readily available for definition of boundary conditions for the simulation.

Facet subsets are entities which are not transferred onto the volume mesh in the meshing process. They are used for definition of meshing settings. Each face subset contains indices of triangles in the surface mesh. Please note that a triangle in the surface mesh can be contained in more than one subset. Facet subsets can be generated by cfSuite, a commercial application developed by Creative Fields, Ltd.

Feature edges are treated as constraints in the meshing process. Surface points where three or more feature edges meet are treated as corners. Feature edges can be generated by the `surfaceFeatureEdges` utility or `cfSuite`.

Figure 2.17 shows a surface mesh with highlighted patches.



Figure 2.17: An example of geometry with patches and subsets.

All sharp features transferred by `cfMesh` must be defined by the user prior to the meshing process. The edges at the border between the two patches, figure 2.18a, and the feature edges are handled as sharp features in the meshing process, see figure 2.18b. Other edges in the triangulation are not constrained.

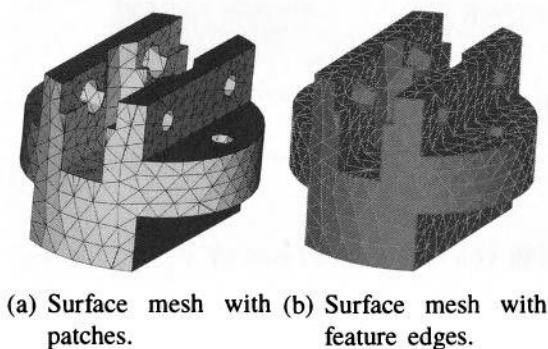


Figure 2.18: Possible ways of capturing feature edges.

The file formats suggested for meshing are: `fms`, `ftr`, and `stl`. In addition, the geometry can be imported in all formats supported by the `surfaceConvert` utility which comes with OpenFOAM. However, the three suggested formats support definition of patches which are transferred onto the volume mesh by default. Other formats can also be used for meshing but they do not support definition of patches in the input geometry and all faces at the boundary of the resulting volume mesh end up in a single patch.

The preferred format for `cfMesh` is `fms`, designed to hold all relevant information for setting up a meshing job. It stores patches, subsets, and feature edges in a single file. In addition, it is the only format which can store all geometric entities into a single file, and the users are strongly encouraged to use it.

Dictionaries and available settings

The meshing process is steered by the settings provided in a `meshDict` dictionary located in the system directory of the case. For parallel meshing using MPI, a `decomposeParDict` located in the system directory of a case is required, and the number of nodes used for the parallel run must match the `numberOfSubdomains` entry in `decomposeParDict`. Other entries in `decomposeParDict` are not required. The resulting volume mesh is written in the `constant/polyMesh` directory. The settings available in `meshDict` will be explained in more detail in the remainder of this section.

The `cfMesh` library requires only two mandatory settings to start a meshing process:

surfaceFile points to a geometry file. The path to the geometry file is relative to the path of the case directory.

maxCellSize represent the default cell size used for the meshing job. It is the maximum cell size generated in the domain.

Refinement settings

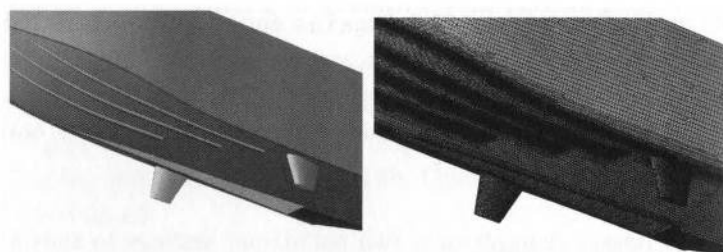
When a uniform cell size is not satisfactory, there are many options for local refinement sources in `cfMesh`.

boundaryCellSize option is used for refinement of cells at the boundary. It is a global option and the requested cell size is applied everywhere at the boundary.

minCellSize is a global option which activates automatic refinement of the mesh template. This option performs refinement in regions where the cells are larger than the estimated feature size. The scalar value provided with this setting specifies the smallest cell size which can be generated by this procedure. This option is use-

ful for quick simulation because it can generate meshes in complex geometry with low user effort. However, if high mesh quality is required, it provides hints where some mesh refinement is needed.

localRefinement allows for local refinement regions at the boundary. It is a dictionary of dictionaries, and each dictionary inside the main **localRefinement** dictionary is named by a patch or facet subset in the geometry which is used for refinement. The requested cell size for an entity is controlled by the **cellSize** keyword and a scalar value, or by specifying **additionalRefinementLevels** keyword and the desired number of refinements relative to the **maxCellSize**.

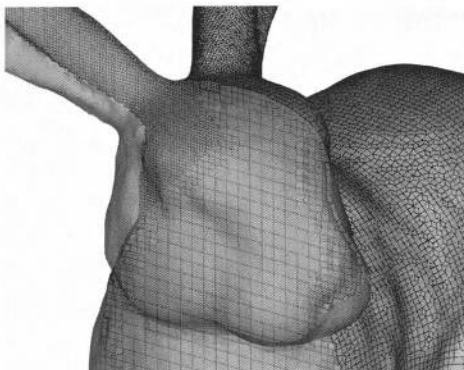


(a) Refinement regions at chines (b) Mesh with local refinement and stabilisers

Figure 2.19: Local refinement via patches/subsets

objectRefinement is used for specifying refinement zones inside the volume. The supported objects which can be used for refinement are: lines, spheres, boxes, and truncated cones. It is specified as a dictionary of dictionaries, where each dictionary inside the **objectRefinement** dictionary represents the name of the object used for refinement.

Meshing workflows implemented in **cfMesh** are based on inside-out meshing, and the meshing process starts by generating the so-called mesh template based on the user-specified cell size. However, if the cell size is locally larger than the geometry feature size it may result with gaps in the geometry being filled by the mesh. On the contrary, the mesh in thin parts of the geometry can be lost if the specified cell size is larger than the local feature size.



(a) Refinement inside a volume of a Stanford bunny.

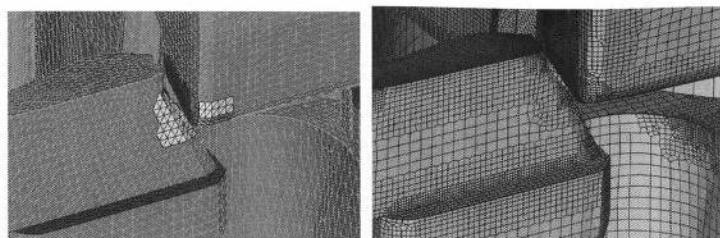
Figure 2.20: Local refinement via primitive objects.

The `keepCellsIntersectingBoundary` option is a global option which ensures that all cells in the template which are intersected by the boundary remain part of the template. By default, all meshing workflows keep only cells in the template which are completely inside the geometry. The `keepCellsIntersectingBoundary` keyword must be followed by either 1 (active) or 0 (inactive). Activation of this option can cause locally connected mesh over a gap, and the problem can be remedied by the `checkForGluedMesh` option which also must be followed by either 1 (active) or 0 (inactive).

The `keepCellsIntersectingPatches` option is an option which preserves cells in the template in the regions specified by the user. It is a dictionary of dictionaries, and each dictionary inside the main dictionary is named by patch or facet subset. This option is not active when the `keepCellsIntersectingBoundary` option is switched on.

The `removeCellsIntersectingPatches` option is an option which removes cells from the template in the regions specified by the user. It is a dictionary of dictionaries, and each dictionary inside the main dictionary is named by a patch or a facet subset. The option is active when the `keepCellsIntersectingBoundary` option is switched on.

Boundary layers in `cfMesh` are extruded from the boundary faces of the volume mesh towards the interior, and cannot be extruded prior to the



(a) Selected regions at the surface mesh. (b) Resolved gap in the volume mesh.

Figure 2.21: Remove cells intersected by the patches/subsets.

meshing process. In addition, their thickness is controlled by the cell size specified at the boundary and the mesh generator tends to produce layers of similar thickness to the cell size. Layers in `cfMesh` can span over multiple patches if they share concave edges or corners with valence greater than three. Furthermore, `cfMesh` never breaks the topology of a boundary layer, and its final geometry depends on the smoothing procedure. All boundary layer settings are provided inside a `boundaryLayers` dictionary. The options are:

nLayers specifies the number of layers which will be generated in the mesh. It is not mandatory. In case it is not specified the meshing workflow generates the default number of layers, which is either one or zero.

thicknessRatio is a ratio between the thickness of the two successive layers. It is not mandatory. The ratio must be greater or equal to 1.

maxFirstLayerThickness ensures that the thickness of the first boundary layer never exceeds the specified value. It is not mandatory.

patchBoundaryLayers setting is a dictionary which is used for specifying local properties of boundary layers for individual patches.

It is possible to specify **nLayers**, **thicknessRatio** and **maxFirstLayerThickness** options for each patch individually within a dictionary with the name equal to the patch name. By default, the number of layers generated at a patch is governed by the global number of layers, or the maximum number of layers specified at any of the patches which form a continuous layer together with the existing patch. `allowDiscontinuity`

option ensures that the number of layers required for a patch shall not spread to other patches in the same layer.

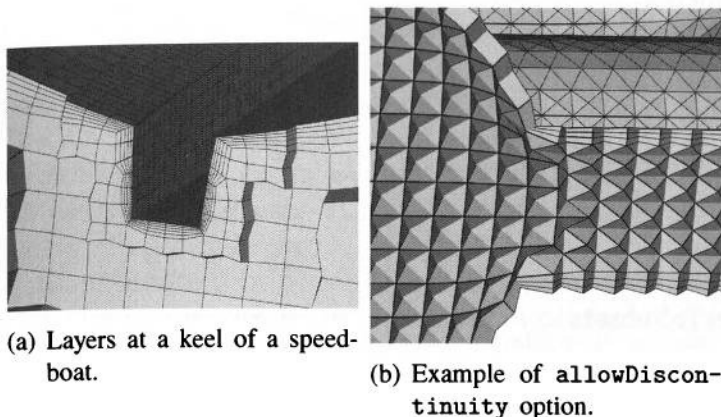


Figure 2.22: Boundary layers.

The settings presented in this section are used for changing of patch names and patch types during the meshing process. The settings are provided inside a `renameBoundary` dictionary with the following options:

newPatchNames is a dictionary inside the `renameBoundary` dictionary.

It contains dictionaries with names of patches which shall be renamed. For each patch it is possible to specify the new name or the new patch type with the settings:

newName keyword is followed by the new name for the given patch. The setting is not mandatory.

type keyword is followed by the new type for the given patch. The setting is not mandatory.

defaultName is a new name for all patches except the ones specified in `newPatchNames` dictionary. The setting is not mandatory.

defaultType sets the new type for all patches except the ones specified in `newPatchNames` directory. The setting is not mandatory.

Various utilities in cfMesh

Currently, following utilities are provided by the cfMesh project:

FLMAToSurface converts geometry from AVL's flma format into the

format readable by `cfMesh`. Cell selections defined in the input file are transferred as facet subsets.

FPMAToMesh is a utility for importing volume meshes from AVL's `fpma` format. Selections defined on the input mesh are transferred as subsets.

copySurfaceParts copies surface facets in a specified facet subset into a new surface mesh.

extrudeEdgesInto2DSurface extrudes edges written as feature edges in the geometry into a ribbon of triangles required for 2D mesh generation. Generated triangles are stored in a single patch.

meshToFPMA converts the mesh into the AVL's `fpma` format.

patchesToSubsets converts patches in the geometry into facet subsets.

preparePar creates processor directories required for MPI parallelisation. The number of processor directories is dependent on the `numberOfSubdomains` specified in `decomposeParDict`.

removeSurfaceFacets is a utility for removing facets from the surface mesh. The facets which shall be removed are given by a patch name or a facet subset.

subsetToPatch creates a patch in the surface mesh consisting of facets in the given facet subset.

surfaceFeatureEdges is used for generating feature edges in the geometry. In case the output is a `fms` file, generated edges are stored as feature edges. Otherwise it generates patches bounded by the selected feature edges.

surfaceGenerateBoundingBox generates a box around the geometry. It does not resolve self-intersections in case the box intersects with the rest of the geometry.

2.3 Mesh Conversion from other Sources

While `blockMesh` and `snappyHexMesh` are powerful mesh generation tools, users may often use third party meshing packages for defining and discretizing a more complex flow domain.

2.3.1 Conversion from Thirdparty Meshing Packages

Many advanced external meshing utilities offer the user additional levels of control during mesh generation. This includes selectable element types,



fitted boundary layer meshes, and length scale control to name a few. Some mesh generators can export directly to a functional OpenFOAM mesh format. Listed below is a compilation of the mesh formats supported for conversion in OpenFOAM 2.2:

- Ansys
- CFX
- Fluent
- GMSH
- Netgen
- Plot3D
- Star-CD
- tetgen
- KIVA

The capabilities of the importing utilities vary strongly, as well as the nomenclature used by them. The Fluent import tool converts *internal boundaries* to *faceSets*, whereas other tools ignore such features completely.

If your particular meshing software is not mentioned in the above list, it is more than likely that it is capable of exporting a mesh into a supported intermediate format.

TIP

The source code for all of the above mentioned conversion utilities are found here: `$WM_PROJECT_DIR/applications/utilities/mesh/conversion/`.

Users also have the option of converting OpenFOAM meshes into Fluent or Star-CD mesh formats using the `foamMeshToFluent` and `foamToStarMesh` utilities. This could be especially useful for exporting meshes generated from the `snappyHexMesh` utility mentioned previously.

The mesh conversion process is typically very straightforward with very little syntax changes between the different conversion utilities. For that reason, only one example will be given and will be based on the `fluentMeshToFoam` conversion utility. To begin the process, copy a tutorial case to a directory of choice, this tutorial is based upon the existing mesh conversion tutorial for the `icoFoam` solver.

```
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/elbow/ ./
```

```
?> mv elbow meshConversionTest
?> cd meshConversionTest
```

Converting the mesh is as simple as running the conversion utility and passing the mesh file as the argument, which must be present in the directory. During conversion the utility will output patch names and mesh statistics to the console. The polyMesh files will be updated accordingly.

```
?> fluentMeshToFoam elbow.msh
```

It is important to keep in mind that the imported mesh is only as good as the exported. In case of the Fluent mesh, 2D meshes are not possible to import, since OpenFOAM only supports three-dimensional meshes.

After the import is complete, the case will need to be updated to reflect the new patch names in the initial and boundary condition files. All existing patches can either be gathered from the output of the import tool, or looked up manually by opening `constant/polyMesh/boundary` with an editor. For this tutorial the U and p fields were pre-configured for this particular mesh.

Scaling the mesh during import is as simple as adding the option and scaling factor to the command. For the sake of this tutorial, the mesh should be scaled down by one order of magnitude.

```
?> fluentMeshToFoam -scale 0.1 elbow.msh
```

When constructing a mesh in many third-party meshing utilities, users can often assign boundary condition types such as inlet, outlet, wall, etc. to each particular patch. The conversion process will attempt to match certain boundary condition formats to a corresponding OpenFOAM format but there is no guarantee of success or accuracy in boundary condition conversion. It is crucial to check that the conversion correctly parsed the flow information. To check this, inspect `constant/polyMesh/boundary` and run `checkMesh` on the newly converted mesh.



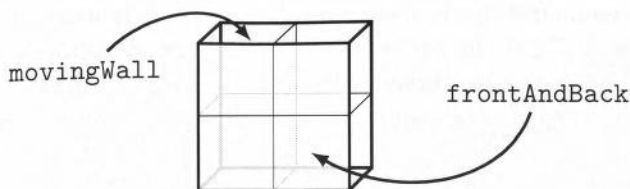


Figure 2.23: Simplified sketch of the cavity example

2.3.2 Converting from 2D to Axisymmetric Meshes

In order to convert a mesh to an axisymmetric one, the following requirements have to be fulfilled. The mesh must already be a *valid* OpenFOAM mesh and it must only be one cell "thick". The latter requirement is valid for all 2D meshes in OpenFOAM. As the cavity example of icoFoam satisfies all these requirements, it is used for this tutorial. It is located at `$FOAM_TUTORIALS/incompressible/icoFoam/cavity`.

With the mesh being shaped as a rectangle and having only one cell, a very basic geometry can be created. In OpenFOAM an axisymmetric mesh has the following properties: the mesh is one cell "thick" and is rotated about an symmetry axis to form a 5° wedge shape. The two angled boundaries of the wedge are considered separate patches of type *wedge*.

TIP

The sources of `makeAxialMesh` are available on the OpenFOAM wiki: http://openfoamwiki.net/index.php/Contrib_MakeAxialMesh. Follow the instructions there to download and compile the utility.

The next steps are to create a copy of the case folder to a working directory of your choice, renaming the directory to avoid any future confusion and creating the 2D base mesh.

```
?> cp -r $FOAM_TUTORIALS/utilities/incompressible/icoFoam/cavity .
?> mv cavity axisSymCavity
?> cd axisSymCavity
?> blockMesh
```

For the axisymmetric mesh, the `movingWall` patch is used as symmetry axis (see figure 2.23). In addition, the single `frontAndBack` patch will be split and act as the two boundaries of the wedge (`frontAndBack_neg` `frontAndBack_pos`). The parameters entered into the command line reflect this:

```
?> makeAxialMesh -axis movingWall -wedge frontAndBack
```

The utility creates a new time directory (in this case 0.005) to store the transformed mesh. If the creation did not work as expected, only this directory needs to be deleted and the basic mesh is restored again. The case directory should now contain the folders shown below:

```
?> ls
0 0.005 constant system
```

At this point the mesh has been warped into a 5° wedge shape, as shown in figure 2.24. However, the faces from the `movingWall` patch are present, however, they are now crushed into faces of near zero face area. `makeAxialMesh` transforms the point positions but does not alter the mesh connectivity. Because of this, the symmetry patch has no faces assigned to it (`nFaces` = 0) and must be removed. Using the `collapseEdges` tool is advisable in this case. It takes two mandatory command line arguments: edge length and merge angle:

```
?> collapseEdges <edge length [m]> <merge angle [degrees]>
```

For many applications an edge length of 1×10^{-8} m and merge angle of 179° will correctly identify and remove the recently collapsed faces. In some instances where the mesh edge length scale is extremely small, a smaller edge length may be required to avoid false positives and the inadvertent removal of valid edges. Executing `collapseEdges` with the parameters as shown works without issues for this example.

```
?> collapseEdges -latestTime 1e-8 179
```

For some final housekeeping it is advisable to remove the now empty patches from the boundary list. Open `constant/polyMesh/boundary` and delete the `movingWall` and `frontAndBack` entries. Note that they



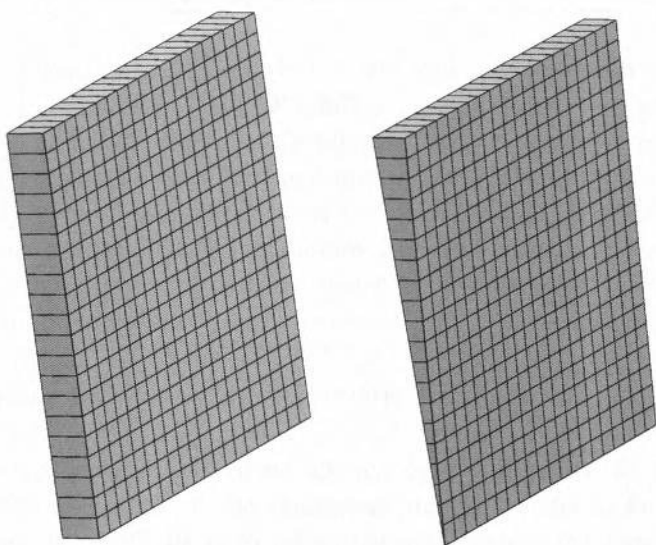


Figure 2.24: The 2D cavity mesh before and after the `makeAxialMesh` wedge transformation

are listed as containing zero faces: `nFaces 0;`. Change the boundary list size to 3 to reflect these two deletions. The boundary file should now look similar to listing 14.

At this point, the `fixedWalls` patch can be split into 3 separate patches using the `autoPatch` utility. This will look at a contiguous patch and try to identify appropriate places to split it based on a given feature angle. In this case, any patch edges that form an angle greater than 30° can be split to form a new patch. This provides more flexibility, when it comes to the assignment of boundary conditions.

```
?> autoPatch -latestTime 30
```

The patches will be renamed after the split. The `-latestTime` flag will only read the latest time step available. Instead of overwriting the time step, the split mesh is stored in yet another time step directory. Finally the mesh should be checked for errors, using the `checkMesh` tool, which should be considered a general rule of best practices: always run

Listing 14 Example boundary file for a wedge shape mesh

```
3
(
  fixedWalls
  {
    type          wall;
    nFaces        60;
    startFace     760;
  }
  frontAndBack_pos
  {
    type          wedge;
    nFaces        400;
    startFace     820;
  }
  frontAndBack_neg
  {
    type          wedge;
    nFaces        400;
    startFace     1220;
  }
)
```

checkMesh when the mesh was changed.



2.4 Mesh Utilities in OpenFOAM

The utility applications (utilities) which deal with mesh operations can be found in the directory `$WM_PROJECT_DIR/applications/utilities/mesh`. The mesh utilities are grouped in the following categories: generation, manipulation, advanced and conversion. Generating the mesh and converting it from different formats into the OpenFOAM format has been described in section 2.2 and section 2.3. This section covers manipulating the mesh as well as advanced operations like mesh refinement once a base mesh has already been generated.

2.4.1 Refining the Mesh by a Specified Criterion

In this example, the mesh refinement application `refineHexMesh` is employed to refine the mesh of the `damBreak` tutorial of the `interFoam` solver. The purpose of this is to refine the area around the initial free surface, where the gradient of the two-phase marker field (α_1) is greater than zero ($\nabla(\alpha_1) > 0$).

To start, a local copy of the `damBreak` tutorial in the working directory of your choice must be generated. All utilities executed by the `Allrun` script must be executed, only the solver is not started. This generates the mesh and initializes the α_1 field.

```
?> cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak .
?> cd damBreak
?> blockMesh
?> setFields
```

The mesh is now generated with the `blockMesh` and the α_1 field is set using the `setFields` pre-processing utility. The `setFields` utility is described section 3.2. The basic calculator utility `foamCalc` can be used to compute and store the gradient of the α_1 field.

```
?> foamCalc magGrad alpha1
```

This will store the cell-centred scalar field of the gradient magnitude in the initial time directory 0 named `magGradalpha1`. To refine the mesh based on the gradient magnitude using the `refineMesh` application,

the configuration dictionary file for this utility must be copied into the system directory of the damBreak case.

```
?> cp $FOAM_APP/utilities/mesh/manipulation/  
    refineMesh/refineMeshDict system/  
?> ls system/  
controlDict      fvSchemes  refineMeshDict  
decomposeParDict fvSolution  setFieldsDict
```

In the configuration now available in the system directory, `refineHexMesh` refines all cells in a certain `cellSet`.

```
// Cells to refine; name of cell set  
set c0;
```

This `cellSet`, when created, will be stored in the `constant/polyMesh` and `refineHexMesh` will use it to refine the cells.

In this case, `topoSet` is used to generate the `cellSet`. This requires the `topoSetDict` to be present in the system directory and configured properly. Therefore an existing one is copied and changed afterwards.

```
?> cp $FOAM_APP/utilities/mesh/manipulation/topoSet/  
    topoSetDict system/
```

The example actions subdictionary of `topoSetDict` must be replaced by the contents of listing 15.

Now the `cellSet` can be generated, based on the definitions in `system/topoSetDict`:

```
?> topoSet  
?> refineHexMesh c0
```

When the mesh is now viewed using Paraview, the area of the free surface should now have additional resolution. Now we will do the same using the `setSet` utility.



Listing 15 Example entries of `addLayersControls`

```

actions
(
{
    name    c0;
    type    cellSet;
    action  new;
    source  fieldToCell;
    sourceInfo
    {
        fieldName magGradalpha1;
        min 20;
        max 100;
    }
}
);

```

2.4.2 transformPoints

In the OpenFOAM mesh format, the only information pertaining to scale and location of the mesh is in the point position vectors. All of the remaining stored mesh information is purely connectivity based as discussed previously. With that said, the mesh size, position and orientation can be altered by transforming the point locations alone. For this purpose, the `transformPoints` mesh utility comes with OpenFOAM. Because this utility is relatively straight forward, only the required syntax is shown. The most often used options when transforming a mesh are the `-rotate`, `-translate` and `-scale` options.

scale scales the points of the mesh in any or all cardinal directions by a specified scalar amount. `-scale '(1.0 1.0 1.0)'` does not change the point locations, while `-scale '(2.0 2.0 2.0)'` doubles the point positions in all directions uniformly. Any non-uniform scaling will stretch or compress your mesh in your given direction(s).

translate moves the mesh by the given vector, effectively adding this vector to every point position vector in the mesh.

rotate rotate the mesh. The rotation is defined by to input vectors. The mesh will undergo the rotation required to orient the first vector with the second. When rotating a mesh, any initial or boundary vector and tensor values can be rotated as well by adding the `-rotateFields` option.

Syntax for these three point transformations are shown below.

```
?> transformPoints -scale '(x y z)'
?> transformPoints -translate '(x y z)'
?> transformPoints -rotateFields -rotate '( (x0 y0 z0) (x1 y1 z1) )'
```

2.4.3 mirrorMesh

Sometimes it is easier to generate a mesh with symmetry planes then performing mirror reflections than meshing the entire geometry in one step. The `mirrorMesh` utility is doing exactly that. All parameters, with regards to the mirroring process itself, are read from a dictionary. This is located in `system/mirrorMeshDict`.

In order to mirror a mesh successfully, the mirroring plane must be planar. For this example a quater mesh is mirrored into a full domain. First, the following solid analysis case must be copied into the directory of choice and renamed to prevent later confusion. The `mirrorMeshDict` needs to be copied from an existing case into the case system directory.

```
?> cp -r $FOAM_TUTORIALS/stressAnalysis/\
    solidDisplacementFoam/plateHole .
?> mv plateHole mirrorMeshExample
?> cd mirrorMeshExample
?> cp -r $FOAM_APP/utilities/mesh/manipulation/\
    mirrorMesh/mirrorMeshDict system/
```

The next step is to define the plane which will act the the mirror-plane. Such a plane can be defined by an origin and a normal vector, in the `mirrorMeshDict` as shown below. Patches about which the reflection is taking place are automatically removed. After this is defined properly, `mirrorMesh` can be executed and the mesh should be checked for errors (see listing 16).

Listing 16 Entries of `mirrorMeshDict` for the first mirroring action

```
pointAndNormalDict
{
    basePoint      (0 0 0);
    normalVector   (0 -1 0);
}
```



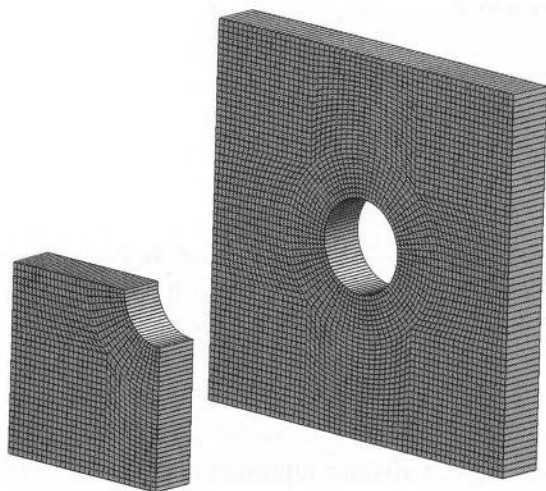


Figure 2.25: A 1/4 mesh before and after mirroring

```
?> mirrorMesh
?> checkMesh
```

This gives a half mesh. For the second mirroring, the dictionary must be changed accordingly, as shown in listing 17.

Listing 17 Entries of `mirrorMeshDict` for the second mirroring action

```
pointAndNormalDict
{
    basePoint      (0 0 0);
    normalVector   (-1 0 0);
}
```

```
?> mirrorMesh
?> checkMesh
```

This results in a full domain mesh, instead of a symmetric fraction as shown in figure.2.25.

2.5 Summary

There is a wide choice of open source applications available for designing the domain geometry, as well as for generating the mesh. On the other side, there are commercial solutions for the same purpose. Geometry definition and mesh generation sometimes involve a tedious workflow until the geometry is reduced to a level acceptable for simulation, and the generated mesh has sufficient quality. For simple cases, e.g. for validation, `blockMesh` is still a preferred mesh generation tool for many users. Automated mesh generation solutions such as `snappyHexMesh`, `cfMesh`, `foamHexMesh`, and `engrid` reduce the complexity and duration in mesh generation, but still involve handling different mesh generation parameters. Mentioned applications are distributed along with their own detailed documentation, so this chapter represents an effort to provide information on the way the mesh in OpenFOAM is composed, as well as additional information on mesh generation with `blockMesh` and `snappyHexMesh` to extend the currently available documentation.

Further reading

Höpken, J. and T. Maric (2013). *Feature handling in snappyHexMesh*.

URL: www.sourceflux.de/blog/snappyhexmesh-snapping-edges.

OpenFOAM User Guide (2013). OpenCFD limited.

Villiers, E. de (2013). *7th OpenFOAM Workshop: snappyHexMesh Training*.

URL: <http://www.openfoamworkshop.org/2012/downloads/Training/EugenedeVilliers/EugenedeVilliers-TrainingSlides.tgz> (visited on 12/2013).



3

OpenFOAM Case Setup

This chapter covers how simulation cases are organized, as well as to define some boundary conditions for a given case. The focus is on providing basic information on a broad range, rather than very detailed information. This will be done in the later chapters.

WARNING

The simulation test cases produced when following the steps outlined in this chapter are available in the example case repository, in the `chapter3` sub-directory.

3.1 The OpenFOAM Case Structure

An OpenFOAM simulation case is a file directory, sub-structured as a set of files organized into sub-directories. Some files are used to configure and control the simulation, others are used to store resulting simulation data. In general, the file-based organization of an OpenFOAM simulation case is relatively straightforward. Using a file-based organization has one more advantage: the possibility to parameterize simulations easily, as the new simulation is nothing more than a copy of a simulation file directory. Further on, standard Unix tools can be used to edit the case files, even in an automated fashion.

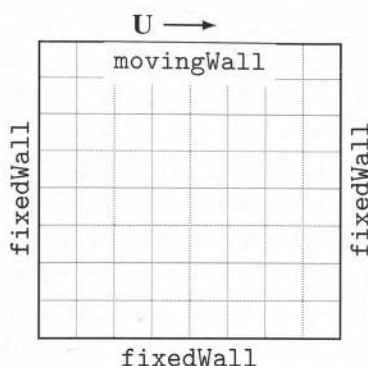


Figure 3.1: Sketch of the cavity

The standard composition of an OpenFOAM case is explained using the cavity tutorial case. This case appears multiple times among the OpenFOAM tutorials, as different solvers can be used to simulate it, which in turn require different settings. The cavity case that is to be run with the `icoFoam` incompressible Navier-Stokes solver, is used in this chapter. Figure 3.1 shows a sketch of the cavity case setup. During the following explanation, other input files are also mentioned, which are not required by `icoFoam`. However, the utilities used during the pre-processing of the case need those files. Generally, with increasing model complexity of the solver, the number of required input dictionaries increases as well.

Listing the sub-directories of the cavity case shows the way the simulation files are organized:

```
?> cd $FOAM_TUTORIALS/incompressible/icoFoam/cavity
?> ls *
0:
  p U
constant:
  polyMesh transportProperties
system:
  controlDict fvSchemes fvSolution
```

The `0`, `constant`, and `system` directories are standard directories of OpenFOAM simulation cases and they have to be present. The `0` directory



holds the initial and boundary conditions applied on the fields. Each field used by the particular solver is represented by a text file, named according to the field. For the cavity tutorial case, the fields used in the simulation are the pressure field p and the velocity field \mathbf{U} . How these fields are defined is shown in section 3.2.

As the simulation progresses, the solver application will write the resulting simulation data to new sub-directories of the case directory. Those directories are the so-called *time step* directories, and they are named based on the simulation time values. They not only contain those fields already defined in the 0/ directory, but also auxiliary fields of the solver, such as the volumetric flux ϕ .

TIP

Although the icoFoam solver starts the simulation with the initial values for the pressure p and the velocity \mathbf{U} fields, the FVM utilises the volumetric flux fields (ϕ) in the process of equation discretization (see chapter 1 for more details). **Finite Volume Method!** (Finite Volume Method!) As a result, the time step directories will hold the volumetric flux field ϕ computed by the solver application.

In addition to the time step directories, other simulation data may get written:

- by the solver application (e.g, the volumetric flux ϕ),
- by a function object running alongside the solver (see chapter 12),
- after the simulation has finished, as a result of post-processing (chapter 4).

The constant directory stores the simulation data that remains constant throughout the simulation. This typically includes the mesh data in the polyMesh sub-directory, as well as various configuration files:

- `transportProperties` - transport properties (described in chapter 11),
- `turbulenceProperties` - turbulence modeling (described in chapter 7),

- `dynamicMeshDict` - dynamic mesh controls (described in chapter 13),
- etc.

Not all aforementioned configuration files are present in the `constant` directory of the cavity case. One that is missing is the `turbulenceProperties` dictionary, which is not required by `icoFoam`. Which additional dictionaries have to be present, depend on the selected solver application. In case a solver is executed in a case directory, without the available necessary input data, the user is prompted with an informative error message. It notifies the user which dictionary, dictionary parameter or even field is missing or ill defined.

TIP

If a solver application uses the dynamic mesh feature, where point positions or mesh topology is changing, a new `polyMesh` folder is written to each time step directory as well. The dynamic mesh feature of OpenFOAM is discussed in chapter 13.

TIP

Configuration files in OpenFOAM are usually referred to as *dictionary files* or even shorter as *dictionaries*. This naming is due to their usage in the source code, which is based on the `IOdictionary` class.

The `system` directory holds all dictionaries relevant to the numerical methodology and the control of iterative solvers. A basic overview of those methods is provided in chapter 1. It may also hold dictionaries used to configure different pre- and post-processing applications such as `setFields`. The most important dictionary contained in the `system` directory, is the `controlDict`. It controls all parameters that relate to the run time of the solver and the frequency with which solution data is written to the case directory. All parameters defined in the `controlDict` are independent of the solver used for the simulation. More information on using the `controlDict` to control the simulation run is presented in section 3.4. A quite extensive discussion of the parameters in the `controlDict` is given in *OpenFOAM User Guide 2013* and some of them are explained in section 3.4.1.



3.2 Boundary Conditions and Initial Conditions

The file-based structure of an OpenFOAM case makes setting both boundary and initial conditions very straightforward. Both are contained in the same files in the 0/ directory. This section covers how boundary and initial conditions are defined. Depending on the field type (scalar, vector, tensor), the respective values have to be defined using a slightly different syntax. This discussion will be limited to case configuration - the numerical and design details of boundary conditions are discussed in chapter 10. Boundary conditions, as their name implies, define the field values at mesh boundaries. Initial conditions cover both the initial boundary field values, as well as internal field values. A sketch for this differentiation is provided by figure 1.8. However, setting initial field conditions is usually concerned with initializing internal field values.

WARNING

More information on the computational mesh can be found in chapter 2.1. The basics of the numerical method have been discussed in chapter 1. Both topics should be known to the reader, to a certain extend.

Before having a closer look at how boundary condition files are defined, the cavity case designed to be simulated with icoFoam has to be copied to a location of choice. To set a basic boundary condition for the cavity simulation case, the simulation case directory needs to be copied and renamed:

```
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity \
    cavityOscillatingU
?> cd cavityOscillatingU
```

Listing the 0/ directory reveals that there are two different fields defined: *p* and *U*. Both files can be edited using any text editor, straight from the command line. The relevant lines of the pressure boundary condition file can be printed to screen using the following command:

```
?> cat 0/p | tail -n 23 | head -21
dimensions      [0 2 -2 0 0 0];
```



```
internalField    uniform 0;

boundaryField
{
    movingWall
    {
        type      zeroGradient;
    }

    fixedWalls
    {
        type      zeroGradient;
    }

    frontAndBack
    {
        type      empty;
    }
}
```

This reveals three top level entries of the boundary condition file: *dimensions*, *internalField* and *boundaryField*. The first one is a set of scalar dimensions (*dimensionSet*), that is used to define the dimensions of the field. Each scalar corresponds to the power of the particular SI unit, as defined in the declaration source file for the dimension set (*dimensionSet.H*):

```
 //- Define an enumeration for the names of the dimension exponents
enum dimensionType
{
    MASS,            // kilogram   kg
    LENGTH,          // metre      m
    TIME,            // second    s
    TEMPERATURE,     // Kelvin    K
    MOLES,           // mole      mol
    CURRENT,         // Ampere    A
    LUMINOUS_INTENSITY  // Candela   Cd
};
```

The next entry is *internalField*, which defines the initial conditions for the field. Note that this does not include the boundaries, which are defined by the last subdictionary: *boundaryField*. In this example, all cell values are set to 0. It is also possible to define initial values on per-cell basis, which in turn requires the user to compose a list with the desired value for each cell. This list must have as many elements as cells are present in the mesh and its composition is explained in *OpenFOAM User Guide 2013*. Lastly, the boundary conditions are defined, inside the *boundaryField*



subdictionary. Boundary conditions must be specified for each patch and each field. Thus the boundary condition for each patch is defined in a subdictionary of the `boundaryField` dictionary. Depending on the type of boundary condition selected, quite a few entries are required.

3.2.1 Setting Boundary Conditions

Already the official release of OpenFOAM comes with a multitude of boundary conditions and a list of all of them is hence omitted. The `foamHelp` command, when executed within a case directory, provides a list of all the boundary conditions available. It accounts for the field type and only shows those boundary conditions that are compatible with the respective field. Some boundary conditions shown in the resulting list are very general, while others are very problem-specific. For the velocity field `U`, it can be called in the following way:

```
?> foamHelp boundary -field U
```

As a small example for defining boundary conditions, the previously copied `cavityOscillatingU` case is used. For this example case the `movingWall` boundary condition for the velocity field `U` is to be changed to a boundary condition that ensures the velocity that varies in time in a sinusoidal fashion (figure 3.1). The `oscillatingFixedValue` boundary condition does exactly this and can be used for this example right out of the box. Somehow the parameters required by `oscillatingFixedValue` must be determined and there are some methods for that. While viewing information via the Doxygen documentation system is the more appropriate approach (chapter 5), having a close look at the header file in the source code of the boundary condition is usually faster. The declaration of the `oscillatingFixedValue` boundary condition in the file must be opened with a text editor. This file is located at:

```
$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived\  
/oscillatingFixedValue/oscillatingFixedValueFvPatchField.H
```

The description section of the large comment section at the top of the file contains an example of the boundary condition can be used:

```
myPatch  
{
```

```
type            oscillatingFixedValue;  
refValue        uniform 5.0;  
offset          0.0;  
amplitude       constant 0.5;  
frequency       constant 10;  
}
```

This example code syntax can be copied directly into the velocity boundary condition of the example case, located at `cavityOscillatingU/0/U`. Just replace the `fixedValue` condition imposed on the `movingWall` patch of the `U` field. The boundary condition parameters are to be set in the following way:

```
movingWall // was myPatch  
{  
  type            oscillatingFixedValue;  
  refValue        uniform (1 0 0);  
  offset          (0 0 0);  
  amplitude       constant 1;  
  frequency       constant 1;  
}
```

WARNING

The scalar values, copied from the boundary condition description in the header file have been modified to vector values, since the boundary condition is to be applied to a vector field.

In addition to the above changes to the `U` field, the simulation time has to be extended so that several velocity oscillation cycles can be visualized. In order to do this, the `system/controlDict` dictionary has to be changed: the entry `endTime` is modified from 0.5 to 3.0. All required adjustments have been done and the changed boundary condition can be tested using the `icoFoam` solver. Before doing so, the mesh must be generated with the `blockMesh` utility:

```
?> blockMesh  
?> icoFoam
```

Visualizing the velocity field can be done using `paraView`. The internal flow pulsates together with the boundary condition. The oscillating velocity of the `movingWall` boundary patch can be computed using the post-processing utility:



```
?> patchAverage U movingWall
```

The utility steps through the timestep directories and outputs the face-area-weighted average velocity magnitude of the movingWall patch.

WARNING

Although a post-processing utility `patchAverage` is used for this example, the description of post-processing utilities is omitted here for better clarity. Details on post-processing are provided in chapter 4.

The `patchAverage` utility provides a console output similar to:

```
Time = 0
Reading volVectorField U
Average of volVectorField over patch movingWall[0] = (1 0 0)

Time = 0.1
Reading volVectorField U
Average of volVectorField over patch movingWall[0] = (1.58779 0 0)

Time = 0.2
Reading volVectorField U
Average of volVectorField over patch movingWall[0] = (1.95106 0 0)

Time = 0.3
Reading volVectorField U
Average of volVectorField over patch movingWall[0] = (1.95106 0 0)

Time = 0.4
Reading volVectorField U
Average of volVectorField over patch movingWall[0] = (1.58779 0 0)

Time = 0.5
Reading volVectorField U
Average of volVectorField over patch movingWall[0] = (1 0 0)
```

The output shows that the value of the x -axis velocity component varies over time, according to the equation outlined in the boundary condition source code.

The boundary conditions are significant parts of the numerical model and are fairly prone to user mistakes. Therefore, careful consideration is always required when selecting boundary conditions. When in doubt of what boundary conditions should be selected for a given problem, a look at the tutorials that come with OpenFOAM is strongly advised. There is

most likely an existing tutorial simulation case that is somewhat similar to a particular problem of interest.

WARNING

When dealing with a new simulation problem, useful information on previously not encountered boundary conditions can be found among the tutorial cases distributed with OpenFOAM. More information on the numerical background, software design and implementation of boundary conditions can be found in chapter 10.

3.2.2 Setting Initial Conditions

In this section a brief overview is provided on how to set the initial condition of a flow field. There is a variety of different tools that initialize the fields according to the user specifications. Chapter 8 provides information on how to develop new pre-processing applications.

TIP

Setting initial field conditions is usually named field *pre-processing*.

Depending on the case and solver, the initial condition may not be important, as it is for example in the cavity tutorial case. In that case, the initial velocity field is set to a uniform vector value of **0**. After the first timestep, the computed flow solution is quite different than the one initially set. For a single-phase incompressible flow, the initial conditions can be considered, more of an initial guess to aid in convergence. Should this guess be excessively far from the appropriate solution, solver divergence may occur.

In other situations, the initial conditions may be extremely important. Compressible flow simulations rely heavily on the initial pressure, temperature, and/or density to properly compute an equation of state. Two-phase flow simulations require a properly pre-processed field that separates the fluid phases.

As stated previously, the cavity example case is relatively tolerant towards the definition of the initial conditions. A case that has special



requirements on the initial conditions is the `damBreak` case, simulated with the `interFoam` solver. The solver `interFoam` is a two-phase flow solver, using an algebraic Volume-of-Fluid method to distinguish between the two immiscible and incompressible fluid phases. For this purpose a new scalar field is introduced: `alpha1`. The gas and the liquid phase in this example will have `alpha1` values of 0 and 1, respectively. For the first example, a water droplet will be added to the `damBreak` case, as shown in figure 3.2.

First, a copy of the `damBreak` tutorial case is made and the case is renamed:

```
?> run
?> cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak \
    damBreakWithDrop
```

Before the `alpha1` field can be initialized, a preparation step needs to be performed. The `0/` directory of the `damBreakWithDrop` case does not contain a file named `alpha1`, only an `alpha1.org` file. The purpose of this is that all fields in this *original* file are initialized using *uniform* values. Later, the `setFields` pre-processing application is used to define initial values for each cell individually, which in turn results in a long list of values: one element for each cell. The `alpha1.org` file is there to be copied to `alpha1`, in order to ensure that all fields are initialized uniformly, without opening a text editor. Thus `alpha1.org` must be copied to `alpha1`:

```
?> cp 0/alpha1.org 0/alpha1
```

TIP

Some tutorials will have field files such as `phi.org` available in the `0` directory. Storing field files in their original state enables the user to go back to the initial field configuration. Alternatively, a version control system might be used for the simulation case directory - more information on this can be found in chapter 6.

Inspecting the current state of the `0/alpha1` field at this point, shows that the entire internal field has a uniform value of 0:

```
internalField uniform 0;
```

The `setFields` utility can be used to create more complex, non-uniform field values. This utility is controlled by the `system/setFieldsDict` dictionary. The template dictionary file for the `setFields` application is stored in the application source directory: `$FOAM_APP/utilities/pre-Processing/setFields`. As this file is fairly long, the contents are not shown. However, it is worthwhile to investigate the contents of `setFieldsDict`, as it stores all the available specifications that can be used by the `setFields` application to pre-process non-uniform fields.

The contents of the `setFieldsDict` for the example test case are shown below, together with the added `sphereToCell` sub-dictionary entry used to initialize a small droplet, i.e. a circle with the `alpha1` value of 1:

```
defaultFieldValues
(
    volScalarFieldValue alpha1 0
);

regions
(
    boxToCell
    {
        box (0 0 -1) (0.1461 0.292 1);
        fieldValues
        (
            volScalarFieldValue alpha1 1
        );
    }
    sphereToCell
    {
        center (0.4 0.4 0);
        radius 0.05;
        fieldValues
        (
            volScalarFieldValue alpha1 1
            volVectorFieldValue U (-1 0 0)
        );
    }
);
```

The `defaultFieldValues` will set the internal field to the provided default value 0, before proceeding to process the `regions` sub-dictionary. The syntax of `setFieldsDict` is mostly self-explanatory for simple



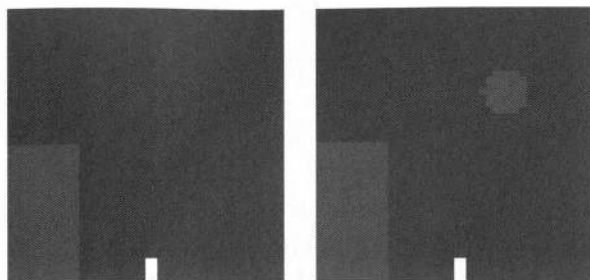


Figure 3.2: A side by side of the initial conditions before and after adding the droplet.

shape-volume selections like those shown above. For a chosen volume, all cells whose cell center is within this volume will have the given fields set accordingly. For example, the `sphereToCell` source selects cells whose centers are within the sphere of a specified center and radius, and sets the field values $\alpha_1 = 1$ and $\mathbf{U} = (-1, 0, 0)$.

In order to pre-process the `alpha1` field and run the simulation, the following steps need to be executed in the command line:

```
?> blockMesh
?> setFields
?> interFoam
```

An illustration of the initial conditions which were changed due to our additions to the `setFieldDict` is shown below in figure 3.2.

WARNING

An interesting exercise might be to modify `setFieldsDict` such that the case is a square and circular gas bubble submerged in a liquid. This would be somewhat of the opposite of the example used in this section: a rectangular slug of liquid and droplet in ambient gas. Radically different results will occur due to the buoyancy effects.

WARNING

If you are interested in quantifying the behavior of the `interFoam` solver, the described test case can be modified so that the computational domain is a square, and a single square is initialized in its center, under zero-gravity conditions. The full description of this validation case and other interesting validation cases are given by J. Brackbill, D. Kothe, and Zemach 1992.

3.3 Discretization Schemes and Solver Control

Choosing a discretization scheme as well as adjusting control parameters for the linear solvers are as important proper handling of boundary conditions. Both general properties are defined by a single dictionary, stored in the `system` directory of the case. Discretization schemes are defined in `system/fvSchemes` and the solver is controlled by the `system/fvSolution` dictionary.

3.3.1 Numerical Schemes (fvSchemes)

From a user perspective, all definitions related to discretization and interpolation are defined in the `system/fvSchemes` dictionary. The settings required differ from solver to solver and are dependent on the formulation of the particular terms of the mathematical model. Discretization and interpolation schemes are used within the framework of the FVM to discretize the terms of the mathematical model. In OpenFOAM, the mathematical model is defined in the solver application, using the Domain Specific Language (DSL). DSL has been developed as the abstraction level the algorithmic implementation became higher and higher with time. In other sources of information regarding OpenFOAM, the OpenFOAM DSL is often referred to as *equation mimicking*. Using algorithms in a higher level of abstraction with equation mimicking allows a very human-readable definition of mathematical models, as well as a trivial modification to the mathematical model. For example, the following source code snippet from the `icoFoam` solver application shows the OpenFOAM implementation of the momentum conservation law equation:



```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);

solve(UEqn == -fvc::grad(p));
```

The different terms of the momentum equation are easily recognized, added, deleted or modified. For each of these terms an equivalent discretization needs to be defined in `system/fvSchemes`.

TIP

A Domain Specific Language (DSL) is developed on purpose for some applications, and for others, it is a natural consequence of software development with a clear separation of abstraction layers. Thinking in terms of discretized equations, matrices, and source terms, and *not* in terms of e.g. iteration loops, variable pointers and functions, represents the foundation of equation mimicking / DSL in OpenFOAM. Separating levels of abstraction is a sign of good software development practices.

The discretization and interpolation schemes are the building blocks of operations listed in the source code above. They convert differential local equations into algebraic equations, with properties averaged in finite volumes. Those terms, that take part in the assembly of the algebraic system are named *implicit terms*. Terms that are explicitly evaluated (located on the r.h.s. of the equation), are named *explicit terms*. Explicit terms are evaluated with the discrete operators from the `fvc` (*finite volume calculus*) namespace, whereas implicit terms are evaluated using the operators from the `fvm` (*finite volume method*) namespace. It is important to memorize this difference, since the discretization scheme defined for a term in the `system/fvSchemes` dictionary file will then be used as default for both `fvc::` and `fvm::` operators in the application code.

To get a list of the supported schemes for a specific term, simply replace the existing scheme by *any word* and execute the solver. The error printed by the solver complains that a scheme by the name chosen on purpose, does not exist. But this is followed by a large list, showing all schemes available. In order to get the information about the scheme parameters

(scheme-specific keywords and values), the user needs to browse through the source files related to the scheme implementation.

The equations implemented in the solver application are defined at *compile-time*. However, the choice of the schemes used to discretize the terms of those equations is performed at *runtime*. This allows the user to modify the way the mathematical model is discretized: choosing different schemes for different cases, by modifying the entries in `system/fvSchemes`.

TIP

Discretization and interpolation schemes are generic algorithms in OpenFOAM. To distinguish between the implicit algorithms (matrix assembly) and the explicit algorithms (source terms), the algorithms are categorized into C++ *namespaces*. A namespace is a programming language construct that is used to avoid name lookup clashes (more details given by Stroustrup 2000)

The `system/fvSchemes` dictionary of the `cavityOscillatingU` case looks like the following:

```
ddtSchemes
{
    default      Euler;
}

gradSchemes
{
    default      Gauss linear;
    grad(p)      Gauss linear;
}

divSchemes
{
    default      none;
    div(phi,U)   Gauss linear;
}

laplacianSchemes
{
    default      none;
    laplacian(nu,U) Gauss linear orthogonal;
    laplacian((1|A(U)),p) Gauss linear orthogonal;
}

interpolationSchemes
{
```



```

    default      linear;
    interpolate(HbyA) linear;
}

snGradSchemes
{
    default      orthogonal;
}

fluxRequired
{
    default      no;
    p            ;
}

```

Every `fvSchemes` dictionary has the same 7 subdictionaries and most likely a default parameter defined at the beginning of each subdictionary. Comparing the code snippet of the moment equation as it is implemented in `icoFoam` to the above listing shows that in this configuration, `div(phi,U)` is discretized using the `linear` method. In the following, an overview of different discretization scheme categories, and characteristics of selected schemes is provided.

WARNING

If the following description of selected schemes is not clear after the first read, it doesn't impact the understanding of the rest of the book. Those interested may return to this part of the book later and read it again alongside chapter 1. Describing all of the available schemes in OpenFOAM is outside of scope for this book, so only a few selected schemes were addressed.

The schemes are sorted into categories that correlate to the terms of the mathematical model as well as the entries in `system/fvSchemes`:

- `ddtSchemes`
- `gradSchemes`
- `divSchemes`
- `laplacianSchemes`
- `interpolationSchemes`
- `snGradSchemes`

ddtSchemes

The **ddtSchemes** are schemes used for temporal discretization. The Euler first-order temporal discretization scheme is set as a default value for transient problems, and it happens to be the scheme used to show the discretization practice of the FVM in chapter 1. Alternative choices of the temporal discretization schemes are:

- **CoEuler**,
- **CrankNicolson**,
- **Euler**,
- **SLTS**,
- **backward**,
- **bounded**,
- **localEuler**, and
- **steadyState**.

Of the available schemes for the temporal discretization, **CoEuler** and **backward** are explained more thoroughly in the following:

CoEuler scheme is a first order temporal discretization scheme that can be used for both implicit and explicit discretization. It automatically adjusts the time step locally so that the local Courant number is limited by a user specified value. The scheme computes a local time step field using the current time step value and a scaling coefficient field computed from the local Courant number field. The reciprocal of the Courant number limited local time step is calculated as

$$\delta t_{fCo}^{-1} = \max(Co_f / \max Co, 1) \delta t^{-1}, \quad (3.1)$$

where the local face centered Courant number is computed from the volumetric flux F , the face area normal vector \mathbf{S}_f , and the distance between the cell centers of the cells connected at the face \mathbf{d} :

$$Co_f = \frac{\|F\|}{\|\mathbf{S}_f\| \|\mathbf{d}\|} \delta t. \quad (3.2)$$

In case the mass flux is provided, the density field needs to be used to compute the local face-centered Courant number using

$$Co_f = \frac{\|F_\rho\|}{\rho_f \|\mathbf{S}_f\| \|\mathbf{d}\|} \delta t, \quad (3.3)$$



where F_ρ is the mass flux stored at the face center. Once the face centered Courant number is computed, the reciprocal of the time step is computed using equation 3.1. For a face-centered field ϕ_f , the reciprocal of the face-centered time step is then used to compute the first-order temporal derivative:

$$\delta t_{fCo}^{-1}(\phi_f^n - \phi_f^o). \quad (3.4)$$

Since the temporal schemes operate mostly on cell centered fields, a cell-centered value for the reciprocal of the time step is computed as a maximum of the face values, i.e.

$$\delta t_c^{-1} = \max_f(\delta t_{fCo}^{-1}), \quad (3.5)$$

which is then used to compute the temporal derivative of the cell-centered field ϕ_c :

$$\delta t_{cCo}^{-1}(\phi_c^n - \phi_c^o). \quad (3.6)$$

This way, using a locally computed field for the reciprocal of the time step δt^{-1} , the temporal derivative is evaluated locally. Local estimation of the temporal derivative based on the local Courant number allows for larger time steps to be applied in those parts of the flow domain, where the Courant numbers are lower. Thus the simulation speed is increased (numerical time is artificially going faster in those parts of the domain, since the flow there is expected to experience a smaller amount of change). Limiting is enforced based on the local Courant number, because by doing so, the numerical stability of the solution is enforced.

backward scheme or Backward Differencing Scheme of second order (BDS2), uses the field values from the current and the two successive old time step fields to assemble a second order convergent temporal derivative. The derivative is computed using a Taylor series expansion starting at the current time, going back two time steps:

$$\phi^o = \phi(t - \delta t) = \phi(t) - \phi'(t)\delta t + \frac{1}{2}\phi''\delta t^2 - \frac{1}{6}\phi'''\delta t^3 + \dots \quad (3.7)$$

$$\phi^{oo} = \phi(t - 2\delta t) = \phi(t) - 2\phi'(t)\delta t + \frac{1}{2}\phi''4\delta t^2 - \frac{1}{6}\phi'''8\delta t^3 + \dots \quad (3.8)$$

Multiplying equation 3.7 with 4 and subtracting equation 3.8 from that, results in the BDS2 temporal discretization of a cell-centered field ϕ_c :

$$\phi'_c \approx \frac{\frac{3}{2}\phi_c - 2\phi_c^o + \frac{1}{2}\phi_c^{oo}}{\delta t}. \quad (3.9)$$

The calculation of the derivative using old values is done in OpenFOAM without requiring the client code to store old field and mesh values. All volumetric and face centered fields have the ability store the values from the old (*o*) and the old-old (*oo*) time step values automatically. So does the mesh itself store information needed for the temporal discretization of higher order (e.g. old and old-old mesh volume fields). The `backward` scheme computes a second order temporal derivative which may be explicitly evaluated, since the *o* and *oo* field values are known. The actual implementation takes into account the possible adjustment of the time step, resulting with the derivative of the cell-centered fields as

$$\phi'_c \approx \frac{c_t\phi_c - 2c_{to} + c_{too}\phi_c^{oo}}{\delta t}, \quad (3.10)$$

where *c* coefficients are defined in the following way:

$$c_t = 1 + \frac{\delta t}{\delta t + \delta t^o}, \quad (3.11)$$

$$c_{too} = \frac{\delta t^2}{\delta t^o(\delta t + \delta t^o)}, \quad (3.12)$$

$$c_{to} = c_t + c_{too}, \quad (3.13)$$

and in case of the two sub-sequent time steps being equal, the *c* coefficients correspond to the discretization using a constant time step, compare equation (3.9).

gradSchemes

The `gradSchemes` determine which gradient evaluation schemes are used terms defined in the solver. There are many examples in practice where the choice of the gradient scheme may lead to a better solution. For example, the gradient is used to compute the surface tension force in two-phase flow simulations, so it plays an important role in flows driven by surface tension. Whenever steep gradients are present, or different



mesh topology is applied (e.g. tetrahedral meshes), a gradient scheme using a wider stencil of cells may provide better solutions.

Available gradient schemes in OpenFOAM are:

- Gauss
- cellLimited
- cellMDLimited
- edgeCellsLeastSquares
- faceLimited
- faceMDLimited
- fourth
- leastSquares
- pointCellsLeastSquares

Three of these schemes are selected and described in the following: Gauss, cellLimited and pointCellLeastSquares.

Gauss is the most often used discretization scheme for the gradient, the divergence (convective), and laplacian (diffusive) terms. It is also described in chapter 1. This scheme expects face centered values in order to compute the cell-centered gradient.

cellLimited extends the functionality of the standard gradient scheme. It computes a limiter for the cell centered gradient value, computed in the standard way and then scales the gradient value with this limiter ($0 < l \leq 1$). The limiter is calculated using the maximal and minimal values (ϕ_{cmax} , ϕ_{cmin}) of a cell centered property ϕ_c , found by indirectly searching through the face adjacent cells as

$$\phi_{cmax} = \max_C(\phi_c), \quad (3.14)$$

$$\phi_{cmin} = \min_C(\phi_c), \quad (3.15)$$

where ϕ is the cell-centered property and C is the set of all face-adjacent cells of a cell c . Once the minimal and maximal values in the face-connected cell stencil are found, the maximal and minimal values are used to compute the value differences using the original cell centered value ϕ_c :

$$\Delta\phi_{max} = \phi_{cmax} - \phi_c, \quad (3.16)$$

$$\Delta\phi_{min} = \phi_{cmin} - \phi_c. \quad (3.17)$$

The differences are then increased with the user specified coefficient ($0 < k < 1$) as

$$\Delta\phi_{max} = \Delta\phi_{max} + \left(\frac{1}{k} - 1\right) (\Delta\phi_{max} - \Delta\phi_{min}) \quad (3.18)$$

and

$$\Delta\phi_{min} = \Delta\phi_{min} - \left(\frac{1}{k} - 1\right) (\Delta\phi_{max} - \Delta\phi_{min}). \quad (3.19)$$

Note that the differences are not increased if the user specified coefficient k is set to 1.

TIP

The equations in this section rely on the cell-to-cell connectivity and they also use cell-based stencils. The actual implementation makes use of the owner-neighbor addressing, for reasons described in chapter 1.

The limiter is initially set to 1, but as it is computed by looping over the mesh faces using h , its value will be updated. This is why the min and max functions are comparing the ratio of the property difference given my minimal and maximal values with the *extrapolated difference given by the gradient*:

$$\Delta(\phi_c)_{grad} = \nabla(\phi)_c \cdot \mathbf{cf}, \quad (3.20)$$

where \mathbf{cf} is the vector connecting the cell and the face center in question. The increased differences compared to the minimal/maximal values are then used to compute the limiter (l):

$$l = \begin{cases} \min(l, \frac{\Delta\phi_{max}}{\Delta(\phi_c)_{grad}}) & : \Delta\phi_{max} < \Delta(\phi_c)_{grad} \\ \min(l, \frac{\Delta(\phi_c)_{grad}}{\Delta\phi_{min}}) & : \Delta\phi_{min} > \Delta(\phi_c)_{grad} \end{cases} \quad (3.21)$$

Determining the final gradient in the cell center, the gradient computed by the standard scheme is scaled with the limiter

$$\nabla(\phi)_c = \nabla(\phi)_c \cdot l. \quad (3.22)$$



pointCellsLeastSquares provides a larger stencil for computing the gradient. It introduces the neighboring cells, that are adjacent to the cell in question, not just across cell faces, but also across points. On an unstructured hexahedral mesh, this stencil would contain 3^3 cells. Although the numerical schemes implemented in OpenFOAM are *second order convergent*, when sharp jumps are present in the flow domain, *absolute accuracy* becomes another very important aspect of the numerical approximation. Using a wider stencil, this scheme provides an estimation of the gradient with lower absolute errors. If a linear Taylor expansion for a property ϕ around the cell center c is used:

$$\phi(\mathbf{x}) = \phi_c - \nabla(\phi)_c \cdot (\mathbf{x} - \mathbf{x}_c) + O(\|\mathbf{x} - \mathbf{x}_c\|^2) \quad (3.23)$$

three values are needed to compute the three unknown variables: the components of the gradient $\nabla(\phi)_c$. The least squares gradient involves expanding the Taylor series (equation 3.23) from the center of the cell where the gradient is computed to neighboring cells. The number of expansions is directly proportional to the size of the cell stencil. For two-dimensional triangular meshes, the gradient can be computed directly from the surrounding three cells from the face-neighbor stencil. However, including more cells in the stencil increases the absolute accuracy of the gradient. When more than three point values are known, the system becomes overdetermined. Consequently the gradient is computed using a minimization of the squared gradient error defined as

$$E = \sum_{cc} w_{cc}^2 E_{cc}^2, \quad (3.24)$$

where cc are the cells of the stencil. w_{cc} is the weighting factor relating the cell in question and the cell of the stencil. Usually an inversed distance between two cells is taken as the weight and E_{cc}^2 is the squared error of the Taylor expansion from the cell for which the gradient is computed to the cell of the stencil:

$$E_{cc} = \frac{1}{2} \nabla \nabla(\phi)_c : (\Delta \mathbf{x} \Delta \mathbf{x}). \quad (3.25)$$

After some algebraic manipulation (Mavriplis 2003), the minimization of the squared error results in a linear algebraic system for the

gradient components. The linear algebraic system size is assembled and solved for each cell, since the system dimensions are 3×3 . Furthermore, the solution is obtained by inverting the coefficient matrix per cell directly since the coefficient matrix will be a symmetric tensor. As a motivation for using this gradient scheme, consider figure 3.3, where the gradient of the volume fraction field is computed using the standard Gauss linear and the new pointCellsLeastSquares gradient scheme. The volume fraction field is set as a circle of radius $R = 2\text{cm}$ centered in the domain of $6 \times 6\text{ cm}$, discretized using a mesh with 30×30 volumes (J. U. Brackbill, D. B. Kothe, and Zemach 1992). The pointCellsLeastSquares computes a more *uniform* gradient of a circular droplet compared to the Gauss linear, since the point neighbors are involved in the gradient calculation. Involvement of point neighbors increases the absolute accuracy for fields with sharp jumps.

WARNING

The Gauss gradient calculation shown for the example field in figure 3.3 is a textbook example of *mesh anisotropy* - the calculation strongly depends on the orientation of faces with respect to the coordinate axes.

WARNING

As an exercise, try computing the gradient of steep explicit functions on different meshes and compare the error convergence for different gradient schemes.

divSchemes

The `divSchemes` are used for discretizing any convective (divergence) term in the mathematical model. Viewing the `system/fvSchemes` dictionary of the `cavityOscillatingU` case shows that the spatial discretization scheme most often encountered for the divergence terms is the Gauss discretization scheme, which uses the Gauss-Ostrogradsky divergence theorem. In chapter 1, the usage of the Gauss theorem was shown by means of the discretization process for the divergence term in equa-



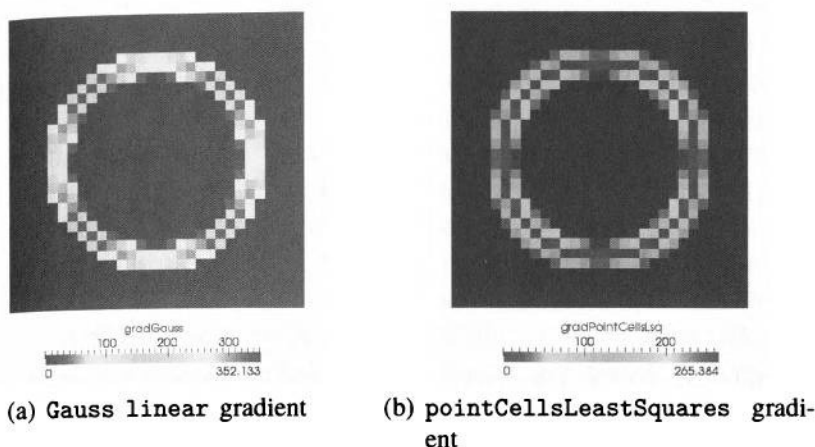


Figure 3.3: Comparing the gradient of a circular volume fraction field

tion (1.5). The discretization scheme presents the basis of the FVM, since it produces a system of algebraic equations, which makes it unlikely to be changed in the configuration file for implicit terms. Additionally, the bounded option may be provided for simulations involving steady-state, or partially converged solutions where $\nabla \cdot (\mathbf{U}) = 0$ is not exactly satisfied, during the iterations of the solution algorithm. In that case, the term $\nabla \cdot (\mathbf{U})$ term is deducted from the coefficient matrix to improve the solution convergence. For example, such correction of the divergence term discretization can be used to increase the convergence rate of the steady state solver `simpleFoam`.

laplacianSchemes

The `laplacianSchemes` subdictionary in the `systems/fvSchemes` dictionary is assembled from `Gauss` discretization, and an `interpolationScheme`. `Gauss` discretization is the base choice for the discretization practice for diffusive (laplacian) terms, with the choice of the interpolation scheme left available to the user.

interpolationSchemes

The `interpolationSchemes` change the way the face centered values are interpolated. Different interpolation schemes may be chosen together with the spatial discretization scheme, such as the Gauss discretization scheme. On the other hand, the values on the faces may be interpolated using a larger set of point-value-pairs. Thus increasing the accuracy of the interpolation, and the Gauss discretization scheme will then be used to discretize the divergence, laplacian, or gradient term using face interpolated values.

OpenFOAM provides a very large variety of choices when it comes to interpolation schemes: `biLinearFit`, `blended`, `clippedLinear`, `CoBlended`, `cubic`, `cubicUpwindFit`, `downwind`, etc. The majority of the available interpolation schemes are used for discretizing the convective term. Applying specific numerical schemes introduces different numerical errors in the convection, especially if the convected property has a jump in the solution domain. For example, it is commonly known the Central Differencing Scheme (CDS, `linear`) introduces numerical instabilities in the solution, and the Upwind Differencing Scheme (UDS, `upwind`) smooths the convected field artificially (Ferziger and Perić 2002, Versteeg and Malalasekera 1996). As a result, different numerical schemes were developed to counter the negative effects of the original schemes, either involving higher order interpolations, or computing the final interpolated value as a combination of values obtained by other interpolation schemes.

WARNING

Those readers that have survived up to this point have surely noticed that there are two aspects that complicate learning of discretization/interpolation practices implemented in OpenFOAM: additional complexity introduced by the unstructured mesh and the necessary background in numerics. However, the modularity of OpenFOAM makes the implementation of the schemes very readable and clear. As a result, an interested learner is able to follow the scheme implementation alongside a book on numerical methods and learn the schemes on his/her own, without a deep understanding of the C++ programming language.



3.3.2 Solver Control (fvSolution)

The discretized formulation of the mathematical model leads to a system of algebraic equations of shape $\mathbf{Ax} = \mathbf{b}$. This system must be solved somehow, which can be done using either direct or iterative methods. Direct methods have very high computational costs, when solving large sparse matrices, as pointed out by Ferziger and Perić 2002. All settings related to the solving process of such matrix equations, as well as pressure-velocity coupling are done in the `system/fvSolution` dictionary. Depending on the solver used, the contents of that file change. For the previously used cavity tutorial of the `icoFoam` solver, the following definitions in the `fvSolution` dictionary are preset:

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
}

PISO
{
    nCorrectors        2;
    nNonOrthogonalCorrectors 0;
    pRefCell            0;
    pRefValue           0;
}
```

This dictionary contains two main subdictionaries: `solvers` and `PISO`. The `solvers` subdictionary needs to be present for all solvers, as it contains the choice and parameters for the various *matrix* solver used. It is important to distinguish the matrix solvers from the solver applications, that implement a certain mathematical model. The second dictionary stores parameters needed by the PISO pressure-velocity cou-

pling algorithm. There are other pressure-velocity coupling algorithms in OpenFOAM, such as SIMPLE and PIMPLE. Each of them requires a dictionary with the particular algorithm name to be present in `fvSolution`.

WARNING

There are two types solvers in OpenFOAM: *solver applications* and *linear solvers*. Solver applications are programs used directly by the end-user for running simulations. Linear solvers are algorithms used to solve linear algebraic equation systems as a part of the simulation solution procedure. Usually, the solver applications are named "solvers" in short, so this term is used for solver applications from this point on.

Linear Solvers

There are various matrix solvers, for symmetric and asymmetric matrices as well as several preconditioners and their working principles were discussed in chapter 1.3.4. The example shown above, uses PCG to solve the pressure equation and the matrix is preconditioned using DIC, which can be used for symmetric matrices. For the momentum equation on the other hand, the asymmetric PBiCG solver is used, in conjunction with a DILU preconditioner.

Regardless of the matrix solver, the following parameters can be defined:

tolerance defines the exit criterion for the solver. This means that if the change from one iteration to the next is below this threshold, the solving process is assumed to be sufficiently converged and stopped. When performing e.g. steady state simulations of a steady problem, the tolerance should be quite small, in order to improve the convergence and accuracy. For transient problems on the other hand, no steady solution can be obtained and hence the tolerance must not be chosen to be very small.

relTol defines the relative tolerance as the exit criterion for the solver. If it is set to some other value than 0, it overrides the tolerance setting. Whereas **tolerance** defines the *absolute* change between two consecutive iterations, **relTol** defines the *relative* change. A



value of 0.01 forces the solver to iterate until a change of 100% is reached, between two consecutive iterations. This comes in handy, when a strongly unsteady system is simulated and the tolerance setting leads to high iteration numbers.

maxIter is an optional parameter and has a default value of 1000. It defines the maximal number of iterations, until the solver is stopped anyway.

Pressure-Velocity Coupling

Depending on the solver application selected, a different pressure-velocity coupling algorithm is implemented in the respective solver. The solver application tries to read the subdictionary in `fvSolution`, depending on the pressure-velocity coupling algorithm implemented.

TIP

For more background information on the various algorithms, the reader is referred to Ferziger and Perić 2002; Versteeg and Malalasekera 1996. There is plenty of information available on the OpenFOAM wiki as well.

There are some parameters that have to be defined anyway:

nNonOrthogonalCorrectors parameter defines a number of internal loop cycles which are used to correct for mesh non-orthogonality. This parameter is necessary when e.g. tetrahedral meshes are used or when local dynamic adaptive mesh refinement is applied on hexahedral meshes. The effects of non-orthogonality to the equation discretization are described in detail by Jasak 1996 and Ferziger and Perić 2002. The internal loop is introduced because the correction is explicit. It needs to be applied multiple times, as each application reduces the errors and does not completely remove them.

pRefPoint/pRefCell Either of these options define the location where the reference pressure is assumed to be located. `pRefPoint` takes a vector in the mesh coordinate system, whereas `pRefCell` takes just the label of the cell where the pressure should be located. For multiphase flows, the point should always be covered by either of the phases and not changing in between. These are only required

if the pressure is not fixed by any boundary condition.
pRefValue defines the reference pressure value and is usually zero.

Other parameters may be required by the particular pressure-velocity coupling algorithm or the solver. The more complex the solver is, the more options are usually required to be present in `fvSolution`. Tutorials for the solvers usually provide sufficient information to get a case running.

3.4 Solver Execution and Run Control

Executing a solver is as easy as issuing any other command under Linux: just type the command name and hit enter. As a general rule of thumb, solvers and any other OpenFOAM utilities should be executed right inside the case directory. Executing a solver from some other directory, requires to pass the `-case` parameter to the solver, followed by the path to the case. For the cavity case, simulated with the `icoFoam` solver, the command looks as such:

```
?> icoFoam
```

Depending on the mesh size, the time step and the total time that is to be simulated this command does take significantly longer, as the usual `ls` or `cp` command. Additionally the information printed to the terminal is massive and gets lost as soon as the terminal is closed. Hence the syntax to execute the solver should be extended:

```
?> nohup icoFoam > log &  
?> tail -f log
```

The `nohup` command instructs the shell to keep the job running, even when the shell window is closed or the user logged out. Rather than printing everything to the screen, the output is piped into a file called `log` and the job is moved into the background. As the job is running in the background and all output is forwarded to the `log` file, the `tail` command is used to basically print the end of the file to the screen. By passing `-f` as a parameter, this is updated until the command is quitted by the user. An other option to keep up to date with the running job is to use the `pyFoamPlotWatcher`, which parses any log file and generates



gnuplot windows from that. These windows are updated automatically and include residual plots, which is handy for monitoring a simulation.

Opposed to manually starting the solver in background and redirecting the output to a log file, the `foamJob` script can be used as well:

```
?> foamJob icoFoam
```

The `foamJob` script is quite powerful as it provides an all in one solution for starting OpenFOAM jobs. Some features of `foamJob` are covered in the next section.

3.4.1 controlDict Configuration

Each case must possess a `system/controlDict` file that defines all the runtime related data. Including the time when to stop the solver, the time step width, the interval and method how time step data is written to the case directory. The content of this dictionary is re-read automatically during the run time of a solver and hence allows for changes, while the simulation is running.

The *OpenFOAM User Guide* (2013) provides an overall introduction to the parameters and parameter combinations, so this information is not repeated here. Two parameters are explained here in more detail, as they are referred to in the rest of the book.

writeControl denotes when data is written to disk. The most popular choices are `timeStep`, `runTime` or `adjustableRunTime`. The actual interval is defined by `writeInterval`, which only takes scalar values. For the sake of simplicity, this interval is named n from now on.

If `timeStep` is chosen every n -th time step is written to disk, whereas choosing `runTime` writes every n -th second to the case directory. The last commonly used option is `adjustableRunTime`, which writes every n -th second to disk but adjusts the time step so that this interval is exactly matched. Hence only nicely named time directories occur in the case folder.

Using the default settings, the amount of data that is written to disk is not limited by OpenFOAM. Especially in cases where a lot of

users access the same storage unit and long run times are common, this fills up the disks in no time.

purgeWrite can be used to circumvent the above mentioned issue with using excessive storage. By default it is 0 and does not limit the amount of time instances written to disk. Changing this to 2 instructs OpenFOAM to keep only the latest 2 time instances on disk and delete the other ones, each time data is written to disk. This option cannot be used with `writeControl` set to `adjustableRunTime`.

In addition to these standard parameters, the `controlDict` also contains custom libraries that are linked to the solvers during runtime as well as calls to function objects. Function objects are covered in chapter 12.

3.4.2 Decomposition and Parallel Execution

Up to this point, all solvers were executed on a single processor. The structure of CFD algorithms makes them viable for data parallelism: the computational domain is divided into multiple parts, and the same task is executed in parallel on each part of the computational domain. Each process communicates with its neighbors and shares relevant data. With modern multicore architectures and HPC clusters, distributing the workload over multiple computing units usually results in an execution speed up in terms of execution time. The decomposition of the computational domain must never influence the numerical properties of the method: consistency, boundedness, stability and conservation. In OpenFOAM, the data parallelism is achieved in a very elegant way, and it is closely tied to the underlying FVM. In effect, the boundary face that was used to elaborate the equation discretization in chapter 1 might as well have been a face of the processor (process) boundary. Before the simulation can be executed in parallel, the computational domain must be divided into as many subdomains as processes are used for the simulation.

TIP

The execution of a solver on a single processor core is often referred to as "serial exeuction" - the solver was executed "in serial".

As an example of parallel execution, the cavity simulation case of the



icoFoam solver is distributed over two cores of the machine the job is started on. This does imply that the machine posses at least 2 cores, otherwise the simulation will take considerably longer than a serial run. In order to make use of the two cores, the data must get distributed between them.

WARNING

Interaction between the interprocess communication (IPC) and the FVM on unstructured meshes in OpenFOAM is a complicated topic and is outside of the scope for this book.

Make a copy of the cavity tutorial for the icoFoam solver:

```
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity $FOAM_RUN/
?> run
?> cd cavity
```

In the next step, a choice is made on the way the computational domain will be decomposed. Various methods exist in OpenFOAM for decomposing the domain. The `scotch` domain decomposition is used for this tutorial. The domain decomposition configuration is stored in the dictionary file `system/decomposeParDict`. The cavity tutorial case does not hold this file by default, though. Just any `decomposeParDict` that is located somewhere in the tutorials of OpenFOAM is fine to copy to the local cavity case.

```
?> cp -r $FOAM_TUTORIALS/multiphase/interFoam/ras/damBreak/\
    system/decomposeParDict $FOAM_RUN/pitzDaily/system
```

The most important lines of `decomposeParDict` are the following:

```
numberOfSubdomains 4;

method                simple;
```

The first line defines how many sub-domains (or processes) will be used, and the second selects the method should used to decompose the domain. Using the `simple` method is generally a bad choice for real world applications, because it splits the domain in spatially equal parts. This happens without optimizing for inter-processor boundaries nor for a load balancing

between the sub-domains. Considering a mesh that is very dense on one side and very coarse on the other, the simple method would slice it in half, producing sub-domains with very uneven distribution of finite volumes. Of course, the more finite volumes the mesh has, more time will be required to finish the simulation. Increased inter-process communication time can severely decrease the efficiency of the parallel execution.

To optimize this, various automatic decomposing methods exist, that e.g. optimize the processor boundaries in order to result with a minimal interprocess communication overhead. By changing the lines in `system/decomposeParDict` as follows, it is ensured that the domain is decomposed in two sub-domains and that the method `scotch` is employed for that:

```
numberOfSubdomains 2;  
method             scotch;
```

Before decomposing the domain, the mesh must be generated. This example uses a `blockMesh` based mesh, which must hence be executed. Once the mesh is generated, the domain decomposition using `decomposePar` must be executed:

```
?> blockMesh  
?> decomposePar
```

Two new directories are generated in the case directory: `processor0` and `processor1`. Each of them contains a sub-domain, including the mesh (in the `processor*/polyMesh` directory) and the fields (in the `processor*/0` directory). The actual command to start the simulation in parallel is a little bit longer than serial command, because MPI needs to be used for that:

```
?> mpirun -np 2 icoFoam -parallel > log
```

This invokes `mpirun` with 2 subprocesses to run `icoFoam` in parallel and store the output in `log` for later assessment. The `-parallel` parameter of `mpirun` is really important because it instructs `mpirun` to run each process with its particular subdomain. If this parameter would be missing, 2 processes would get started but both employ the entire domain, which is not only redundant but also a waste of computational power.



When the simulation has finished, the subdomains need to be reconstructed into one single domain. For this the `reconstructPar` tool is used that - by default - takes all time instances from each processor directory and reconstructs them. Only the last time step can be selected, when `reconstructPar` is called with the optional argument `-latestTime`. It instructs `reconstructPar` to only reconstruct the latest time. This comes in handy when the mesh is pretty large and the reconstruction takes a lot of time and disk space.

Though one might observe a good scaling of the execution time with the number of processors that are used, it is usually a bad idea to go below a certain total cells to subdomain ratio. The processes slow themselves down and the bottleneck then is not the available processor power but the communication between processes. Executing a simulation in parallel results with a speedup s given as

$$s = \frac{t_s}{t_p}, \quad (3.26)$$

where t_s and t_p is the serial and parallel execution time, respectively. A linear (ideal) speedup, is equal to the number of used processes N_p . Usually, because of the interprocess communication, or bottlenecks which are local to a sub-domain (aforementioned local calculations), speedup will have smaller value than the number of processes $s < N$. With simulations that involve larger meshes, the difference between N_p and the evaluated s will be larger. It may happen, however, that *super-linear* speedup is observed, where $s > N_p$: the reason behind is usually specific to the algorithm and the architecture on which the simulation is executed.

3.5 Summary

In this chapter we took the next step of the CFD workflow into the case setup and simulation running phase. This entails setting the initial and boundary field values for the fields involved in the simulation, which can be done with various utilities. Discretization and interpolation schemes are a critical component of the finite volume method in OpenFOAM. They can be selected by the user at the start, or during the simulation. In OpenFOAM there are many interpolation and discretization schemes available, so we mentioned only a few of them - the full description of

all schemes falls outside the scope for this book. We walked through the numerical foundations for some schemes, and their clean software design allows the interested reader to learn how scheme works without a deep understanding of the C++ programming language. Finally, we showed how a user can execute a flow solver in both serial and parallel modes.

Further reading

- Brackbill, J. U., D. B. Kothe, and C. Zemach (1992). “A continuum method for modeling surface tension”. In: *J. Comput. Phys.* 100.2, pp. 335–354. ISSN: 0021-9991.
- Brackbill, J.U, D.B Kothe, and C. Zemach (1992). “A continuum method for modeling surface tension”. In: *Journal of Computational Physics* 100.2, pp. 335–354.
- Ferziger, J. H. and M. Perić (2002). *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer.
- Jasak (1996). “Error Analysis and Estimatio for the Finite Volume Method with Applications to Fluid Flows”. PhD thesis. Imperial College of Science.
- Mavriplis, Dimitri J. (2003). *Revisiting the Least-squares Procedure for Gradient Reconstruction on Unstructured Meshes*. Tech. rep. National Institute of Aerospace, Hampton, Virginia.
- OpenFOAM User Guide* (2013). OpenCFD limited.
- Stroustrup, Bjarne (2000). *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Versteeg, H. K. and W. Malalasekera (1996). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method Approach*. Prentice Hall.



4

Post-Processing, Visualization and Data Sampling

This chapter covers the basics of post-processing, visualization and data sampling. Not only OpenFOAM tools are discussed, but also paraView is used for various tasks, as well as runtime data sampling.

4.1 Post-processing

The post-processing step of a CFD analysis is the task where useful information is extracted from the computed CFD solutions. Extracted data may take the form of images, animations, plots, statistics, and the like. Thankfully, OpenFOAM is equipped with a multitude of post-processing tools that accomodate the needs of most users. The source code of the stock post-processing tools is stored in various categorized subdirectories in the `$FOAM_UTILITIES/postProcessing` directory:

```
?> ls $FOAM_UTILITIES/postProcessing
dataConversion    lagrangian        patch              stressField
wall foamCalc     miscellaneous      sampling           velocityField
turbulence graphics    noise             scalarField
```

While some post-processing calculations may not be directly computable with an existing utility, a combination of calculations using stock utilities may get the job done. Similarly to other OpenFOAM utilities, post-processing applications operate on all existing time directories in a simulation case, by default. Times and time ranges can be selected explicitly by passing the `-time` parameter and the last time can be selected by the `-latestTime` parameter. Before going into detail of example workflow scenarios, some existing post-processing tools are described in the following. While it is beyond the scope of this book to describe all of them, a few selected applications are covered that should be valuable to many users.

foamCalc

The first selected tool is `foamCalc` and it can be used to perform various calculations on existing flow fields. All calculated results are stored as a new field in the respective time directory. The general syntax is:

```
?> foamCalc <operation> <field> <arguments>
```

In the syntax above, `<operation>` defines the type of calculation to perform, `<field>` denotes the field to operate on, and `<arguments>` supply operation specific controls. The various arithmetic and discrete calculus operations that can be performed using `foamCalc` are defined within the `Foam::calcTypes` namespace. Additional arguments, that are common among the majority of OpenFOAM utilities, such as `-latestTime` and `-case` can also be used with this utility. All existing `calcTypes` are:

```
?> ls $FOAM_SRC/postProcessing/foamCalcFunctions
Make          basic          calcType      field
```

These existing `calcTypes` provide the following arithmetic or field calculus operations:

addSubtract Adds or subtracts a value or another field from an existing field. Three examples for addition and subtraction (using fields or uniform constants) are illustrated here:

```
?> foamCalc addSubtract T add -value 5000
?> foamCalc addSubtract T add -value 5000 -resultName Tnew
?> foamCalc addSubtract T subtract -field T0 -resultName Tnew
```



The first example adds 5000 to field T, which overwrites the previous values in this field. To write a new field instead of overwriting the existing one, the `-resultName` argument can be provided. It takes the desired new field name as a parameter, as shown in the second example. The third example subtracts a field T0 from field T and stores the result in Tnew.

components separates the components of a vector or tensor field into separate volume scalar fields. For example, the velocity field U can be separated into three `volScalarFields` (Ux, Uy and Uz) using the following syntax:

```
?> foamCalc components U
```

div computes the divergence of a vector or tensor field and writes the result to a new scalar or vector field respectively. A numerical scheme for the divergence operator (for example, `div(U)`) must be present in the `fvSchemes` dictionary, in order to be executed. The divergence of the velocity field U for the latest time can be computed by

```
?> foamCalc div U
```

And the result is stored as a new `volScalarField` named `divU`, in the particular time step directory.

interpolate interpolates cell centered values to face centers and stores them as a surface field. As a matter of fact, this does not work for face centered fields. For example, a `volVectorField` can be interpolated to faces and stored as a `surfaceVectorField`. Things are similar for scalar and tensor rank fields. The following command interpolates a temperature field T to the face and saves it as `interpolateT`, which is of type `surfaceScalarField`:

```
?> foamCalc interpolate T
```

mag is used to compute the magnitude of a field. For scalar values, the absolute value is computed and for vectorial values, the magnitude is computed. The result is stored in a new field that consists of the old field name with a prepended `mag`. For example, computing the velocity magnitude of the velocity field U can be achieved by executing the following:

```
?> foamCalc mag U
```

magGrad and **magSqr** compute the magnitude of the gradient and the magnitude squared of a field respectively. Both routines write the

computed result as a new scalar field that is named according to the pattern 'magGrad' or 'magSqr' followed by the original field name.

```
?> foamCalc magGrad U
?> foamCalc magSqr U
```

randomise adds a random value to the chosen field within a specified preturbation. Unlike the other operations, the *randomise* calcType is selective with regards to the field type. It works with **vector**, **sphericalTensor**, **symmTensor** and **tensor** fields. Each component of each field value gets its own random value. Below is an example of adding a random preturbation with a maximum value of ± 10 to each component of the velocity field **U**.

```
?> foamCalc randomise 10 U
```

yPlusRAS and yPlusLES

For simulations that employ turbulence modelling, the y^+ value is an important value used to verify if the near wall flow is resolved sufficiently and that resolution is in correct range for that turbulence model. More information on what this value means and how it is calculated can be gathered from chapter 7. The source code for the y^+ post-processing tools is located here:

```
?> ls $FOAM_UTILITIES/postProcessing/wall
wallGradU      wallHeatFlux      wallShearStress
yPlusLES       yPlusRAS
```

The y^+ value is calculated after the simulation is finished, using one of two tools: one for Reynolds Averaged Navier Stokes (RANS) and one for Large Eddy Simulation (LES) based simulations. Both possess the same set of OpenFOAM standard command line arguments, such as **-latestTime** and **-case**. The y^+ values are stored in a new **volScalarField** called **yPlus**:

```
> yPlusRAS # Execute for RANS
> yPlusLES # Execute for LES
```

The y^+ field is calculated only on boundary faces, which are of type **wall**, leaving the remaining cell centered values zero. Besides the freshly written field, both **yPlus** tools print some valuable information to the



screen. This information contains the coefficients used for the particular turbulence model as well as the minimum, maximum and average y^+ value for all wall boundary patches.

patchAverage and patchIntegrate

These post-processing tools can be used to calculate averages and integrals over a patch, respectively. The source code for both tools is located in the patch subdirectory of the postProcessing utilities.

```
?> ls $FOAM_UTILITIES/postProcessing/patch
patchAverage      patchIntegrate
```

patchAverage calculates the arithmetic mean $\bar{\phi}_f$ of a scalar ϕ , weighted by the magnitude of the surface area normal vector $|\mathbf{S}_f|$:

$$\bar{\phi}_f = \frac{\sum_f \phi_f |\mathbf{S}_f|}{\sum_f |\mathbf{S}_f|} \quad (4.1)$$

patchIntegrate computes the integral value for a field on a patch. For this computation, two different approaches are employed: Using the surface area normal vector \mathbf{S}_f and its magnitude $|\mathbf{S}_f|$. Both results are printed to the console, resulting in one vectorial and one scalar result, respectively.

patchAverage can *only* handle a volScalarField, whereas **patchIntegrate** will also handle a surfaceScalarField as input. Both commands require the same set of arguments, when called from the command-line:

```
?> patchAverage <field> <patch>
?> patchIntegrate <field> <patch>
```

In the above code snippet, <field> and <patch> represent the field and patch to operate on, respectively. The results are not stored anywhere in the case directory, but only printed to the terminal. Though they can be stored, by piping the output of the particular utility to a log file:

```
?> patchAverage <field> <patch> > patchAveResults.txt
?> patchIntegrate <field> <patch> > patchIntegrateResults.txt
```

To illustrate the usage of `patchAverage`, the pressure field `p` is averaged over a patch named `pipe`. In order to minimize the output, the calculation is limited to the last time step, which is $t = 50$ s. The resulting terminal output is included.

```
?> patchAverage -latestTime p pipe
Time = 50
Reading volScalarField p
Average of p over patch pipe[2] = 14.3067

End
```

vorticity

The `vorticity` utility calculates the vorticity field ω , using the velocity field `U` and writes the result to a `volVectorField` named `vorticity`. The vorticity of a velocity field represents the local magnitude and direction of rotation in the flow and is defined in equation 4.2. Performing this operation is also known as taking the *curl* of a field. The final output of this utility is the computed vorticity field, the magnitude of vorticity field, and the maximum and minimum vorticity magnitudes in the domain.

$$\omega = \left(\frac{\partial U_z}{\partial y} - \frac{\partial U_y}{\partial z}, \frac{\partial U_x}{\partial z} - \frac{\partial U_z}{\partial x}, \frac{\partial U_y}{\partial x} - \frac{\partial U_x}{\partial y} \right) \quad (4.2)$$

The source code for this post-processing utility can be found in `$FOAM_UTILITIES/postProcessing/velocityField/vorticity`

probeLocations

If field data needs to be probed at certain locations during post-processing, `probeLocations` is the tool of choice. Unlike the previously discussed post-processing tools, this tool requires an input dictionary: `system/probesDict`. It stores two lists, one containing the names of the fields to probe, and one for the locations in space to probe the field values from.

To sample the pressure field `p` and velocity field `U` at two points $[0, 0, 0]$ and $[1, 1, 1]$, the `probesDict` is configured as shown below:




```

fields
(
    p
    U
);

probeLocations
(
    (0 0 0)
    (1 1 1)
);

```

If not stated otherwise, the fields are probed at all existing times with the output files located in a nested subdirectory inside the case directory. The first folder is named **probes** and the subfolder indicates the first time step the data was sampled from. For example, all pressure data gathered by the probing can be found in the file **probes/0/p**. The data is arranged in a tabulated manner with the time being stored in the first column, followed by the extracted field value. This format can then be processed using plotting utilities such as **gnuplot** or **python/matplotlib**. An example of the results of a probing of the pressure field is shown below.

```

# x      0      1
# y      0      1
# z      0      1
# Time
0      0      0
10     3.2323  2.2242

```

Example case studies

As an example to show possible applications for the post-processing tools, a two-dimensional NACA0012 hydrofoil is selected. The hydrofoil is deeply submerged and placed in a domain with a homogeneous inflow. The resulting Reynolds-number is $Re = 10^6$. With the chord length of the NACA profile being $c = 1\text{m}$ and a kinematic viscosity of $\nu = 10^{-6} \frac{\text{m}^2}{\text{s}}$ this gives a freestream velocity of $v = 1\text{m/s}$.

The case can be found in the repository under **chapter4/naca** and should be copied into the user directory:

```

?> cp -r chapter4/naca $FOAM_RUN
?> cd $FOAM_RUN/naca

```

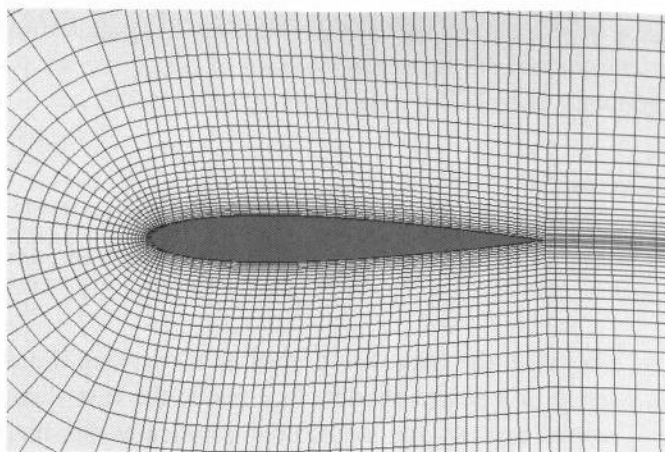


Figure 4.1: Sideview of the NACA0012 profile

To generate the results, the `Allrun` script provided with the tutorial must be executed. After the simulation is finished, the distribution of y^+ over the surface of the NACA profile is inspected. In order to do so, `yPlusRAS` has to be executed in the case directory and the resulting `yPlus` field must be visualized on the surface using `paraView`.

Although `yPlusRAS` prints the minimum, maximum and average y^+ values to the screen, the average value can be recomputed differently, based upon the `yPlus` field. Using `patchAverage` does exactly this:

```
?> patchAverage yPlus FOIL
```

The output is not immediately plot-friendly, as there is some output overhead and superfluous text elements. By using `grep` to look only for floating point numbers at the end of a line, the averages can be filtered out of the `patchAverage` command output. For later processing, they can be piped into a text file `yPlusAverage`. This can be performed by running the command shown below:

```
?> patchAverage yPlus FOIL | \
    grep -o -E '[0-9]*\.[0-9]+' > yPlusAverage
```

Another practical post-processing task is to calculate the integral and

average wall-shear stress in the x direction. The shear stress itself can be computed by `wallShearStress`, which writes a new `volVectorField` to each time step directory. By simply chaining the respective commands, the average `wallShearStress` can be calculated, without the need to use `paraView`:

```
?> wallShearStress
?> foamCalc components wallShearStress
?> patchAverage wallShearStress FOIL | \
    grep -o -E '(-|+)?[0-9]*\.[0-9]+' > stressXAverage
```

4.2 Data Sampling

The post-processing utility `sample` provides an easy to use and powerful way to extract simulation data. While there are many visualization applications that produce attractive imagery, `sample` is better suited for the less visually appealing yet arguably more important role of quantitative analysis. For example, instead of visually estimating the boundary layer thickness from a velocity magnitude representation, it can be extracted from the raw or interpolated velocity data.

In general, `sample` is used to extract and produce 1D or 2D representations of data. Different output formats are supported, as well as different geometrical sampling entities. The application is configured by the `sampleDict`, and one example `sampleDict` is provided alongside with the application source code:

```
?> ls $FOAM_APP/utilities/postProcessing/sampling/sample
Make sample.C sample.dep sampleDict
```

Opening the `sampleDict` in a text editor shows all available configuration options for the `sample` utility. There is a large number of options available, and the provided `sampleDict` is very well documented. The options are described in a clear fashion. `sample` can handle a multitude of sampling parameters: field names, output formats, mesh sets, interpolation schemes, and surfaces.

Regardless of the variety of sampling parameters, `sample` always handles the data sampling process in the same way, regardless of the user's choice of a parameter sub-set. A field to be sampled is chosen by provid-

ing the field name within the `fields` word list. A sub-set of the mesh (`sets` sub-dictionary) or a geometrical entity (`surfaces` sub-dictionary) is used to locate the data sampling points. In case when the data sampling points do not coincide with the mesh points that hold the field data (e.g. cell centers or face centers), the data is *interpolated* using different interpolation schemes (`interpolationScheme` parameter). The interpolated data is then stored in the case, in a specified output format (`setFormat` parameter).

An example of a 1D data extraction is to define a line which intersects the flow domain and sample the velocity field along this line. This can be used to sample e.g. the velocity profile, as usually done for the `cavity` case. This extracted profile can then be compared to other datasets, using any preferred plotting utility.

Another example application of `sample` is to extract boundary field values on large simulation cases. Instead of trying to open up the entire simulation case in `paraView`, `sample` can be used to extract only the values on the boundary patch in question. This localized approach to post-processing using `sample` can drastically reduce the required computational resources, depending on the size of the datasets.

Any simulation case can be used to show how to sample the simulation data using the `sample` utility. For that purpose, the two-dimensional rising bubble test case is selected, available in the `chapter4/risingBubble2D` sub-directory of the example case repository. In order to use `sample` successfully on that case, the simulation has to be executed:

```
?> blockMesh
?> setFields
?> interFoam
```

Sections that follow cover examples of using `sample` and they all involve manipulating the `sampleDict` dictionary file.

4.2.1 Sampling along a Line

In this example the width of the 2D bubble is examined after it has settled on the top wall of the simulation domain. The sampling line



WARNING

Before proceeding with the examples, open the `sampleDict` provided with the sample source code in a text editor of your choice, in order to see all the possible sampling configuration options.

will sample the `alpha1` field at time $t = 7.0s$ along a line which crosses the bubble. A `sampleDict` configured to sample along a line is shown below:

```
setFormat raw;
interpolationScheme cellPoint;
fields
(
    alpha1
    p
    U
);
sets
(
    alpha1Line
    {
        type      uniform;
        axis      distance;
        start      (-0.001 1.88 0.005);
        end        (2.001 1.88 0.005);
        nPoints    250;
    }
);
```

The `setFormat` option changes the format of the data written to file and the `interpolationScheme` option dictates what type (if any) of value interpolation occurs, before data is mapped to the sample line. All fields to be sampled need to be listed in the `fields` list - `alpha1` field in this case. The `sets` subdictionary contains a listing of all of the sample lines that are extracted.

The `type` entry of the `alpha1Line` sub-dictionary defines how the sampled data is distributed along the line - in this case, `uniform` point distribution is used. The `axis` parameter determines how to write the point coordinates - `distance` results in a parametric output, as the coordinate is a distance along the line, starting at the first line point. There are other options available for the `axis` parameter, such as `xyz` where the absolute position vector of the sample point will be written as the first column of

the data file. Next, for a line sample, three-dimensional `start` and `end` points need to be provided. In this setting, `nPoints` determines the number of sample points.

WARNING

Note that the sample line position is slightly adjusted to be away from the boundary. In general, the sample line is not to be co-planar with mesh faces, as it prevents `sample` from determining which cells the line is intersecting.

Once the `sampleDict` has been configured, `sample` can be executed for $t = 7$ s:

```
?> sample -time 7
```

A new directory is created, holding the following files:

```
?> ls postProcessing/sets/7  
alpha1Line_alpha1_p.xy alpha1Line_U.xy
```

The scalar fields `alpha1` and `p` are stored in the same file, while the sampled vector velocity field is stored in a separate file. `alpha1` field values can be visualized from the stored data, using a plotting tool and should lead to a figure similar to figure 4.2.

4.2.2 Sampling on a Plane

Sampling along a plane is especially useful for large 3D cases which are big enough to require a long time to transfer the simulation data across a network connection. The process for setting up the `sampleDict` for a plane is very similar to setting up line sampling. The `surfaceFormat` entry needs to be configured and a list of sample planes needs to be provided inside the `surfaces` sub-dictionary. Similar to line sampling, having the sample plane co-planar with mesh faces should be avoided. The `interpolationScheme` set previously is again used to interpolate the cell centered flow data onto the plane. An example `sampleDict` for this kind of setup looks like this:



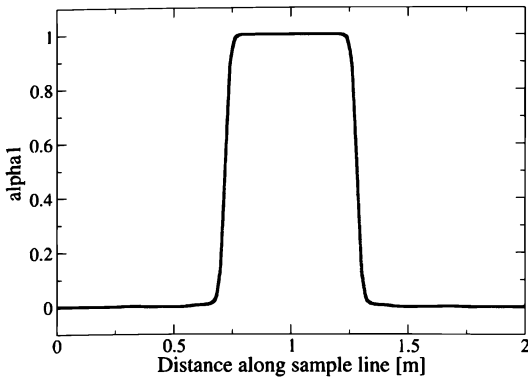


Figure 4.2: A sample of the alpha1 field along our defined line

TIP

If the `-time 7` option is omitted when executing `sample`, every timestep is sampled resulting in the corresponding XY data being interpolated and saved.

```
interpolationScheme cellPoint;

fields
(
    alpha1
    P
    U
);

surfaceFormat vtk;

surfaces
(
    // Sampling a plane
    constantPlane
    {
        type          plane;
        basePoint      (1.0 1.0 0.005);
        normalVector    (0.0 0.0 1.0);
    }
)
```

The VTK output format is chosen for the surface, and the surface type is set to plane in the `constantPlane` sub-directory. The plane is defined using a point (`basePoint`) and a plane normal vector (`normalVector`).

In order to generate the Visualization Toolkit (VTK) planar surface and extract the `alpha1` data, `sample` must only be executed on the final time directory of the `chapter4/risingBubble2D` case:

```
?> sample -time 7
```

A `surfaces/7` sub-directory is created in the `postProcessing` folder and it contains the sampled data:

```
?> ls postProcessing/surfaces/7  
alpha1_fluidInterface.vtk p_fluidInterface.vtk U_fluidInterface.vtk
```

Obviously, as many VTK surfaces are stored as fields are sampled. A surface stored in a VTK format can be visualized by opening it directly in `paraView`, through the `Open` dialogue. In figure 4.3, the sampled plane surface is shown with a sampled `alpha1` field. Since the `risingBubble2D` case is two-dimensional, the result of this sampling example is the same as when the cut filter is used in `paraView`.

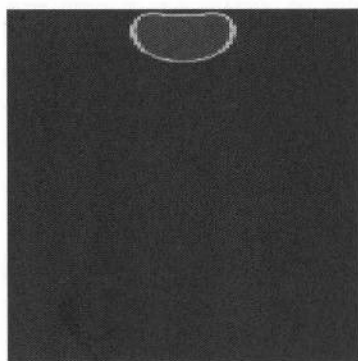


Figure 4.3: A sample of the `alpha1` field on a plane

4.2.3 Generating and Interpolating to an Iso-surface

In addition to extracting data, the `sample` utility can produce iso-surfaces from existing flow data. In this example, `sample` is used to generate



an iso-surface representing the gas-liquid interface and interpolate the pressure field onto it. The `isoSurface` type is placed into the `surfaces` sub-dictionary of `sampleDict`:

```
interpolationScheme cellPoint;

fields
(
    alpha1
    p
    U
);

surfaceFormat vtk;

surfaces
(
    // Sampling an iso-surface
    fluidInterface
    {
        type            isoSurface;
        isoField         alpha1;
        isoValue         0.5;
        interpolate      true;
    }
);
```

In order to sample the iso-surface, `sample` is to be run on for $t = 7$ s:

```
?> sample -time 7
```

In figure 4.4, the `alpha1 = 0.5` iso-surface is shown with the interpolated pressure acting on the bubble.

4.2.4 Boundary Patch Sampling

The capabilities of `sample` are not limited to creating sample planes and sample lines. An entire patch can be extracted, with any desired boundary flow field values mapped to it. Even though a rather small two-dimensional simulation case is used to illustrate this process, patch sampling is better suited for large cases, where loading the entire flow domain is not efficient. In this example, the top wall patch top of the `rising-Bubble2D` example case is extracted and the pressure field acting upon it is visualized. Using the appropriately configured `sampleDict`:



Figure 4.4: An illustration of an iso-surface of $\alpha_1 = 0.5$ colored by pressure p

```
interpolationScheme cellPoint;

fields
(
    alpha1
    p
    U
);

surfaceFormat vtk;

surfaces
(
    // Extracting a patch
    walls_constant
    {
        type          patch;
        patches        ( top );
    }
);
```

the sampling is performed on the final time step of the `risingBubble2D` example simulation case:

```
?> sample -time 7
```

The `sample` utility generates the `postProcessing/surfaces/7` sub-directory in the simulation case directory, with the following contents:



```
?> ls postProcessing/surfaces/7/
alpha1_topPatch.vtk p_topPatch.vtk U_topPatch.vtk
```

One patch VTK file saved per sampled field listed in the fields list.

4.2.5 Sampling Multiple Sets and Surfaces

The `sample` utility allows sampling of different sets and surfaces, thus `sampleDict` stores lists of such elements. A working `sampleDict` configuration file is prepared in the `risingBubble2D` simulation case directory. It stores the configuration for all the examples described in this section. When additional functionalities are required from `sample`, the `sampleDict` dictionary provided with the application source code should be the first place to look.

4.3 Visualization

To visualize the results of the simulation, the visualization application `paraView` will be used. `paraView` is an advanced open-source application used for visualizing field data, and there is a lot of information available on its use at paraview.org/Wiki/ParaView. In order to avoid repetition, only the minimum amount of information required to visualize the results of the OpenFOAM simulations in `paraView` is provided in this section.

`paraView` allows the user to execute a series of filters on the visualized data. The filters support different operations, such as: calculation of streamlines, visualizing vector fields as glyphs, calculating iso-contour surfaces, just to name a few. Some of the presented OpenFOAM post-processing tools can be replicated within `paraView` as well, such as sampling data from points and along lines.

As of version 3.14, `paraView` has a native reader for OpenFOAM data and can since then be used directly to display such data. The detour using `paraFoam` is still working, but not required anymore. In order to make use of the native reader, a file that ends on `.foam` needs to be present in the case directory. Usually, this file is named in the same way as the simulation case directory, but the choice is of no importance.

In the following, the `risingBubble2D` example case for `interFoam` is

selected, to introduce some of the most basic working principles of paraView. This case can be found in the example repository under `chapter4/risingBubble2D`. First of all the simulation must be performed, to generate the data which can be visualized:

```
?> blockMesh
?> setFields
?> interFoam # wait a little bit
?> touch risingBubble2D.foam
```

The last line is important in the context of this section. It generates an empty file with the "foam" suffix, required by paraView to open the OpenFOAM case data. Finally, paraView is used to open the case files and it is started as a background process, leaving the current terminal accessible for further commands.

```
?> paraview risingBubble2D.foam &
```

Once paraView is started, the `risingBubble2D.foam` file should appear, as shown in figure 4.5. Otherwise the OpenFOAM case can be opened by pointing the paraView file browser to the `risingBubble2D.foam` file, via the *Open* dialogue. On the left side of the paraView GUI, options for choosing parts of the mesh and different fields to be read are displayed to the user. This window is named *Properties*, and if it is closed, it can be reactivated using the *View* drop-down menu. The *Properties* view is shown in figure 4.5.

By default, the internal mesh and all the cell centered fields of an OpenFOAM case are chosen for visualization. The fields and the mesh are read when the selection is accepted by clicking on the *Apply* button in the *Properties* window. The choice of the visualized field can be made using the panel above the *Properties* window and the *Pipeline* browser, shown in figure 4.6.

Initially, the mesh is color with a solid color. Clicking on the *Solid* color tab in the panel shown in figure 4.6, produces a drop-down menu with all available fields. The `alpha1` field in the initial time step is shown in figure 4.7.

Manipulation of fields using filters is a very straightforward process in



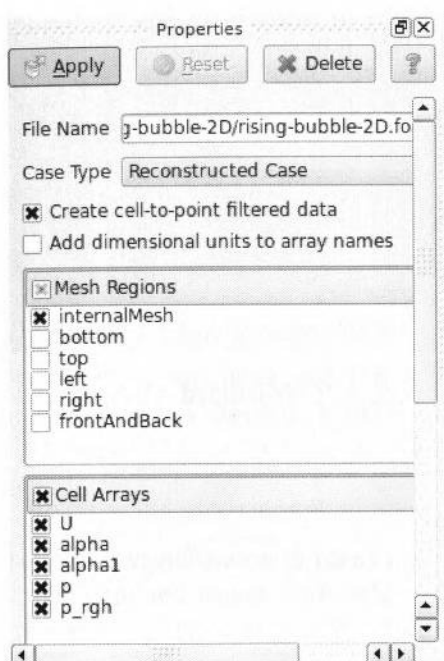


Figure 4.5: paraView Properties window



Figure 4.6: paraView field and display form selection

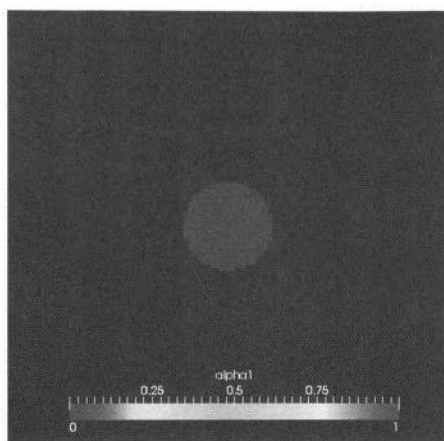


Figure 4.7: Visualized alpha1 field

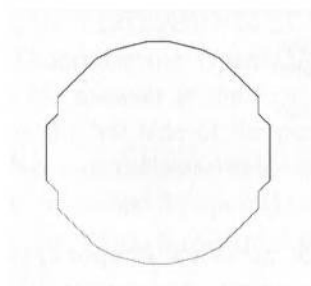


Figure 4.8: Visualized 0.5 iso-contour of the alpha1 field

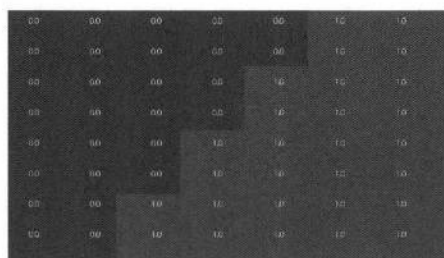


Figure 4.9: Visualized values of the alpha1 field

paraView. Most often used filters are shown as buttons at the bottom part of the main panel shown in figure 4.6. Other filters are available in the **Filters** roll-down menu on the main panel. Documentation covering the use of different filters is extensive, including the paraView community wiki page (paraview.org/Wiki/ParaView), and the official documentation.

One often used Filter is the Contour filter for visualizing the interface between two fluids. The Contour filter computes an iso-surface of a prescribed value, based on a scalar field. In this case the scalar field is α_1 . This operation is in fact fairly similar to the `isoSurface` sampling utility, described in section 4.2. The filter can be selected from the main menu **Filters|Common|Contour**. The scalar field α_1 must be selected for the Contour by parameter and 0.5 for the contour value. When all settings are defined as desired, a click on **Apply** shows the result in the right 3D view. Finally the interface line in the XY plane should be visible, similar to figure 4.8.

An interesting aspect of the visualization in paraView is the ability to enumerate cell and/or point centered values with field data using the **Selection Inspector**. To start, make sure that the "rising-bubble-2D.foam" is selected in the Pipeline Browser. Click on **View->Selection Inspector** and within the Selection Inspector window, and check the **Invert Selection** checkbox. You can decrease the opacity of the selection to 0 in the Display Style sub-window, to prevent it from obscuring the value numbers shown in the cells. To display the values of the α_1 field in each cell, select the **Cell Label** tab of the Display Style sub-window and choose α_1 from the Label Mode drop-down menu. In the Cell Label the format of the shown values can be specified, e.g. "%0.1f" would be a floating point format with a single decimal digit. Figure 4.9 shows a zoomed-in detail of the α_1 field with the displayed cell-centred values.

Extracting values from the top patch can be done using paraView as opposed to the `sample` utility which was discussed in Section 4.2. In order to sample the patch, the boundary mesh and not just the internal mesh needs to be read in paraView. To do so, select the top patch in the Mesh Regions part of the Properties window as shown in Figure 4.10. The top patch is then isolated in the pipeline from the

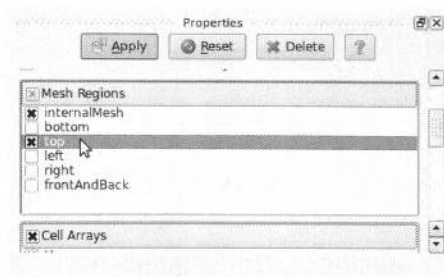


Figure 4.10: Reading the top boundary mesh patch.

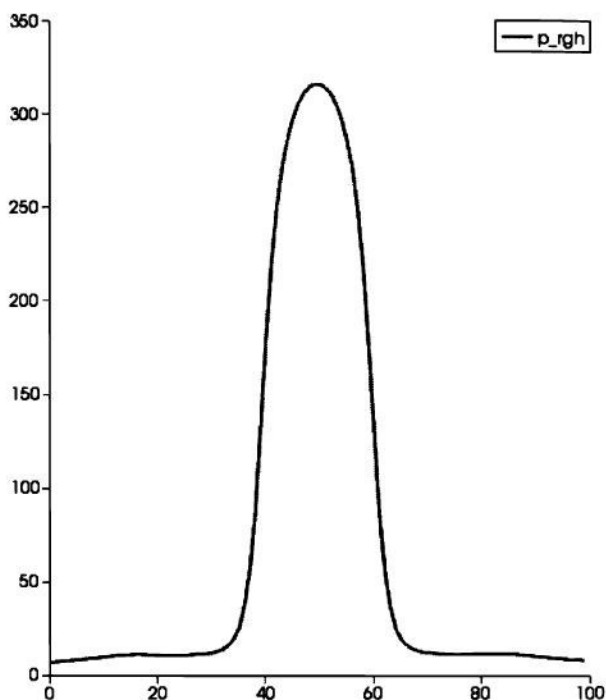


Figure 4.11: Dynamic pressure at the top boundary mesh patch at 7^s simulation time.



rest of the case mesh by selecting it within the Patches branch using the Filter->Alphabetical->Extract Block filter. To plot the data of this patch directly without interpolation, select the extracted block in the pipeline browser and choose Filters->Data Analysis->Plot Data. In the display window (View->Display), select the dynamic pressure p_{rgh} and skip to the last time step of the simulation. The resulting diagram is shown in Figure 4.11.

In this section, the absolute minimal aspects of the workflow with the paraView visualization application are provided. For further details, please consult the material available on the Internet, as well as the official documentation.



Part II

**Programming with
OpenFOAM**

5

Design Overview of the OpenFOAM Library

The FOAM part of the OpenFOAM name is given as an acronym for Field Operation and Manipulation. As noted in the previous chapters, the involved operations and manipulations of fields are quite complex. They involve representing physical properties as tensor sets (called fields in CFD) mapped to representations of the flow domain in forms of computational finite volume meshes. Explicit and implicit changes of such discrete fields are performed. Field changes are described by PDE and their numerical solution usually involves assembling of large and sparse linear systems of algebraic equations. Sometimes the PDE are strongly coupled or a single equation needs to be solved as a pair of coupled equations. In the case of coupled equations, the coupling needs to be taken into account by a numerical algorithm like the pressure-velocity coupling algorithms or a block-matrix coupled solution procedure. Solving the actual linear algebraic equation systems for new field values is performed using iterative solution algorithms because of usually large matrix sizes.

Translating all the aforementioned aspects of the FVM into a software framework is only possible if the chosen programming language supports abstraction of complex concepts. In the context of OpenFOAM these concepts include: fields, finite volume meshes, algorithms, matrices, matrix storage formats, dictionaries, etc. Even this short list shows how

elements occupy different *levels of abstraction*. For example, the field and the mesh concepts are on a higher level of abstraction, than a dictionary data structure. Abstraction allows the programmer to effectively build software elements which model the behavior of complex concepts in the area of interest by allowing him/her to concentrate not exclusively, but more strongly on the implemented concept. This way, the programmer does not have to hold an image of an entire software system in his mind, but jump from one concept to another. Well modeled concepts are those that are strongly cohesive and loosely coupled with one another. This enables them to be easily extensible and replaceable and the software system more modular. For example, an important concept in CFD is a finite volume mesh that is used to discretize the flow domain.

The mesh will hold various geometrical and topological data, as already described in sections 1.3 and 2.1. Additionally, different (often complex) functions are needed to operate on that data. As an example of abstraction in OpenFOAM and the C++ programming language, both the data and the related functions of the finite volume mesh are *encapsulated* into a *class* `fvMesh`. This allows the programmer to *think in terms of a mesh*, and not bother with all the details involving the data structures and functions that build it. Such high-level of abstraction in thinking allows the programmer to write algorithms that use the entire mesh as one of their arguments, which makes the algorithm interface much easier to understand. Otherwise, algorithms working with specific sub-elements of the mesh would have dozens of arguments, as is often the case in procedural programming. Without abstraction, a very large number of global variables would also be present and operated on by various algorithms (routines) which makes it very difficult to determine the program flow.

WARNING

The first sections of this chapter cover an overview of the software design of OpenFOAM. We assume that you are familiar with terms such as: class, paradigm, functional programming, etc. OpenFOAM is a software framework much like any other, so learning software development is a requirement when learning how to program with OpenFOAM.

Programming in a higher level of abstraction is supported by the C++ programming language. It is a multi-paradigm language that supports



procedural, object oriented, generic and recently even the *functional* programming paradigm. Even though the code is human readable, the language still maintains very high computational efficiency and supports a very wide variety of hardware platforms. These aspects make the C++ language a common choice for scientific and computational software development, and thus for OpenFOAM as well.

The best way to understand how different parts of OpenFOAM are designed and interact with each other is through browsing the source code. In order make this easier for the user, both the official and extend OpenFOAM release provide support for generating HyperText Markup Language (HTML) documentation. Generating HTML documentation is performed by via the Doxygen documentation system.

5.1 Generating Local Documentation using Doxygen

In Part II of this book, important classes and algorithms of OpenFOAM are described. Different tutorials are also covered, that describe how to extend the OpenFOAM framework using existing classes and algorithms. It is impossible, however, to cover each and every part of such a large software framework in a single document. The generated Doxygen documentation allows the reader to quickly find information on the class in question. The Doxygen documentation system generates HTML documentation which can be browsed using a web browser, with the advantage of viewing class and collaboration diagrams with linked elements. This makes it especially helpful when interactions between classes and algorithms are investigated. Following links in a HTML browser to a base class may be more effective for most readers compared to browsing through source code using a text editor. Alternatively, an Integrated Development Environment (IDE) might be used when working with OpenFOAM with support for browsing through the source code. To start learning OpenFOAM, we recommend a text editor and the Doxygen generated documentation. Setting up an IDE to work with OpenFOAM might be a complex task in itself for a novice OpenFOAM user.

The settings for the Doxygen documentation system can be found in

`$WM_PROJECT_DIR/doc/Doxyfile`

configuration file.

Often, Unified Modeling Language (UML) is used to describe the design of a software platform. To generate the UML compliant local documentation using Doxygen, UML related tags in the Doxyfile need to be switched on. The configuration of the UML documentation allows hiding and showing private attributes and member functions and other details.

In order to generate the documentation, Allwmake needs to be executed in the

`$WM_PROJECT_DIR/doc`

folder. Once Doxygen has finished generating the documentation, the

`$WM_PROJECT_DIR/doc/Doxygen/html/index.html`

file can be viewed with a web browser. This page is the starting page of the local generated HTML help of the OpenFOAM installation.

EXERCISE

Using only the source code find out what kind of a class may be used with the `tmp<Type>` smart pointer.

Hint: insert "tmp" into the Doxygen search box, click on a first class used with tmp you see in the list, then examine its base classes while keeping tmp.H open in a browser tab.

5.2 Parts of OpenFOAM encountered during Simulations

This section will address some parts of OpenFOAM that are encountered by the user who is running simulations. In Part II, chapters 2, 3 and 4, the prerequisites for this section are covered.

Setting the initial and boundary conditions in chapter 3 has already shown



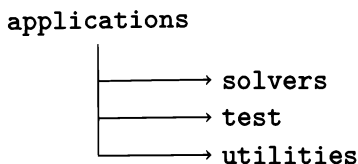


Figure 5.1: Application categories in OpenFOAM.

that the user has considerable flexibility at his or her disposal. Without compiling any additional code, the boundary conditions of the simulation case as well as various control parameters of the simulation can be modified. Input/output operations of simple alphanumeric readable parameters are not complex or difficult to implement. However, choosing a boundary condition is different: an object of a specific class is chosen and instantiated at *runtime* based on a read parameter (boundary condition type) defined by a user. Running simulations puts the user directly or indirectly into contact with the following parts of OpenFOAM: the executable applications (solvers, pre-processing utilities, post-processing utilities), the configuration system (dictionary files), the boundary conditions and the numerical operations (choosing discretization schemes).

5.2.1 Applications

The executable applications are programs that are run by the user in the command line or via a GUI. They belong to what is usually called 'client code', as they are executable programs that make use of various libraries of OpenFOAM. Executable applications (short: applications) are organized in a directory structure as shown in figure 5.1. The applications folder can be easily accessed by executing the alias `app` in the command line, or by switching to the applications directory:

```
?> cd $FOAM_APP
```

The modular design and high level of abstraction in OpenFOAM allows the user to easily build mathematical models. Different solution algorithms for systems of coupled partial differential equations can also be implemented in a straightforward way using OpenFOAM's high level DSL. As a result, a very wide choice of solver applications has become available over time. The solver applications are categorized in groups, as

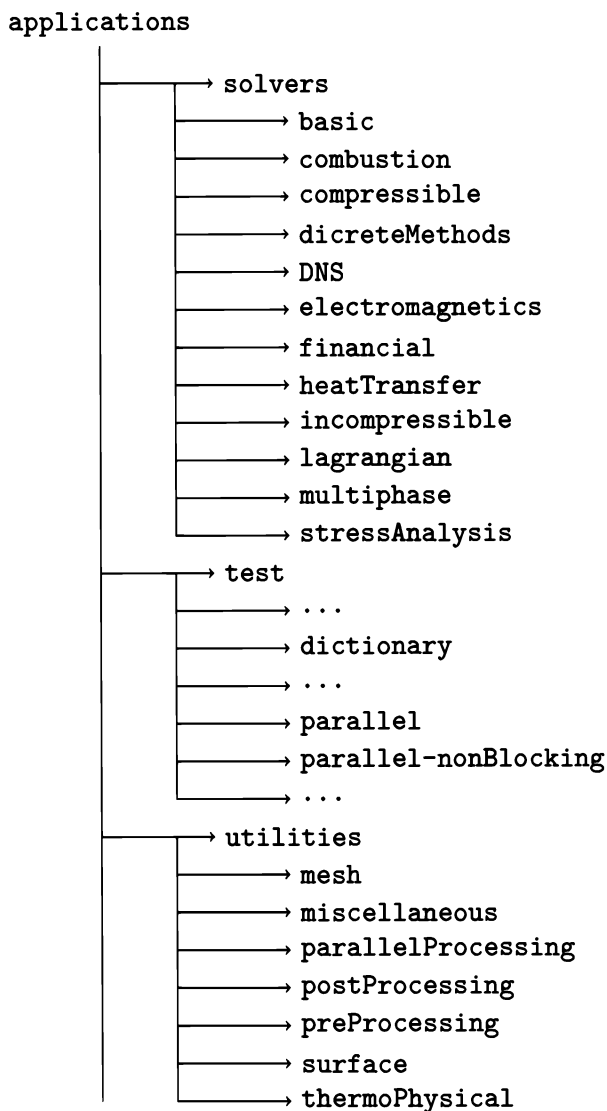


Figure 5.2: Application sub-directories.



shown in figure 5.2.

There are also many testing applications to be found in the `test` sub-directory. For example, the `dictionary` test application tests the main features of the `dictionary` class, while the `parallel` and `parallel-nonBlocking` application implement the code used for abstracting parallel communication in OpenFOAM. Viewing test application source code can be very beneficial in cases when examining the library source code is not enough to understand how the class/algorithm works.

TIP

The test applications are a very useful resource of information regarding specific classes and algorithms. They show how the classes and algorithms are meant to be used.

5.2.2 Configuration System

The configuration system is composed of configuration (dictionary) files. The dictionary file consists of categorized input data which is used for constructing an associative data structure called *dictionary* (hash table). This data structure is used often in OpenFOAM to relate (map) *keys* to *values*.

The various dictionary files found in the simulation case directory contain different parameters used for configuring: boundary conditions, interpolation schemes, gradient schemes, numerical solvers, etc. Choosing different elements as well as initializing them properly is performed at runtime after starting an executable application. The process of choosing types at runtime is called Runtime Selection (RTS) and it employs quite a few software design patterns and language idioms. The RTS process is quite complex, and its description is outside the scope of this chapter. At this point it is only important to remember that it represents the main building block of the *flexibility in use* of OpenFOAM.

Apart from defining types, the dictionary-based configuration system allows setting different physical parameters of the simulation.

For example, any user defined class in OpenFOAM can be made *runtime*

TIP

The configuration system itself does not exist in a form of an abstraction, implemented as a single class. The system is supported by the functionality of the dictionary and IOdictionary classes, Input/Output (IO) file streams, the RTS mechanism, as well as with the design of classes such as the geometric fields and the Time class.

selectable. Additionally, the class attributes can be modified during the simulation - as soon as the dictionary file is modified, the file can be re-read and the class attributes are set to the new values.

TIP

The parameters defined in dictionaries are not changed in the executable application *as soon as they are modified*. Usually, the change is applied *at the beginning of the next time step*, since the checking of modified configuration files is performed by the Foam::Time class.

5.2.3 Boundary Conditions

Boundary conditions are applied on discrete fields, as described in section 1.3, and are implemented as a separate class hierarchy. Because they are implemented as classes, the RTS mechanism allows the user to configure the boundary condition types and parameters as dictionary entries at runtime. Boundary conditions are covered in the chapter 10 so they are not fully addressed in this chapter.

5.2.4 Numerical Operations

The numerical operations are used by the solver and are responsible for equation discretization component of the simulation. The user will be in contact with them when he/she modifies the `system/fvSchemes` dictionary file in the simulation case directory in order to choose a different kind of discretization practice, interpolation scheme, or similar parameter. Different numerical operations have different properties and choosing them requires experience in CFD and also, ideally, in numerical mathematics.



The part of OpenFOAM responsible for numerical operations of the FVM can be found executing the alias `foamfv` in the console, or switching to the appropriate directory:

```
?> cd $FOAM_SRC/finiteVolume
```

Numerical operations are comprised of *interpolation schemes* which are then used by the *discrete differential operators* to discretize the model equations.

The discrete differential operators used often in CFD are: divergence (∇), gradient (∇), laplacian ($\nabla \cdot \nabla$) and curl ($\nabla \times$). They are either explicitly or implicitly evaluated. Explicit operators result with new fields as results. Implicit operators are used to assemble *coefficient matrices* - they discretize the equation terms of the mathematical model. The discrete differential operators are, on the other hand, implemented as function templates, parametrized by the geometric tensor field parameters. This generic implementation of the differential operators makes it straightforward to write mathematical models under the same function names which are used to differentiate tensor fields of various ranks. Since the operators are implemented as functions, assembling a different mathematical model is trivial: a different sequence of function calls results with a different model. OpenFOAM has both implicit and explicit discrete operators that are named the same: e.g. the explicit divergence and the implicit divergence. To avoid ambiguity in the name lookup process, the operators are categorized under two *c++ namespaces* and accessed using fully qualified names (`fvc::` as explicit, `fvm::` as implicit).

The discrete operators are generic algorithms templated for a tensor parameter, using a class trait system to determine the resulting tensor rank. Standard choice of discretization in OpenFOAM is based on the divergence theorem, and the operator calculation is delegated to the discretization scheme. Those interested in developing their own discretization schemes may want to examine the files `fvmDiv.C` and `convectionScheme.C`. Both files contain implementations that show how the divergence term approaches the equation discretization. Source code listing 18 holds the implementation of the code that delegates the divergence calculation from the operator to the convection scheme. Source code listing 18 holds the implementation of the code that delegates the divergence

Listing 18 Convection scheme divergence operation.

```
return fv::convectionScheme<Type>::New
(
    vf.mesh(),
    flux,
    vf.mesh().divScheme(name)
)().fvmDiv(flux, vf);
```

calculation from the operator to the convection scheme. The schemes can be runtime selectable, but they must comply to the interface prescribed by `convectionScheme.H`. Such flexible design can be summarized with the following way:

- the operators delegate the discretization to the schemes,
- schemes compose a runtime selectable hierarchy,
- to write a new convection scheme one must only inherit from `convectionScheme` and add `RTS`,
- once the convection scheme is implemented *none of the solver application code needs to be modified*.

Implementing the operators as function templates and binding them to schemes has many advantages and it prepares the framework for easy extensions without introducing modifications to the existing code. Describing in full detail how the discretization mechanism is implemented lies outside of scope for this book.

The source code which implements other discrete operators can be found in the directory `$FOAM_SRC/finiteVolume/finiteVolume` in the sub-directories `fvc` and `fvm`. The directory `fvc` stores implementations of *explicit discrete operators*, which *result with computed fields*. The directory `fvm` stores the implementation of *implicit discrete operators*, which result in *coefficient matrix assembly* of the algebraic system of equations.

The actions are performed by the solver application, the interaction with the solver code is done when the user modifies the solver in order to make it function in a different way. Recently, the official OpenFOAM release has been extended to take into account solver modifications without requiring



TIP

Whenever there is an `fvc::` (finite volume calculus) operator in the code, the result will be a field. When the `fvm::` operator is encountered, the result will be a coefficient matrix.

the user to modify the solver code. Good examples of these modifications are algorithmic modifications (equations block-coupling) or model modifications (conservative vs. non-conservative discretization, source terms) that can be controlled by the user via dictionary files. More information on solver applications and how to program new solvers is covered in chapter 9. When standard solvers are used to obtain simulation results, their code will typically not be modified by the user.

A high level of abstraction in OpenFOAM allows the user to write new solvers and solution algorithms very quickly. The high abstraction level of OpenFOAM can be nearly be used as a CFD programming language (DSL) - also referred to as *equation mimicking* (Jasak, Jemcov, and Tuković 2007). Take, for example, a mathematical model for a scalar transport of a scalar property T :

$$\frac{\partial T}{\partial t} + \nabla \cdot (T\mathbf{U}) + \nabla \cdot (k\nabla T) = S \quad (5.1)$$

Equation (5.1) describes the transport of a field T composed of a passive advection with the velocity \mathbf{U} , diffusion with the diffusion coefficient field k together with a source term S . Discretization operators, being function templates, can operate on fields of different tensors, and thus on a scalar field they produce the following model equation in OpenFOAM:

$$\text{ddt}(\text{phi}) + \text{fvm}::\text{div}(\text{phi}, T) + \text{fvm}::\text{laplacian}(k, T) = \text{fvc}::\text{Sp}(T)$$

The code describing the model is consisted of discrete operators `ddt`, `div`, `laplacian` and `Sp`.

Interpolation schemes also belong to the numerical operations in OpenFOAM. The most often used interpolation schemes are built upon owner-neighbour addressing of the unstructured mesh with the interpolation resulting in face-centered values. From this, a tree like class hierarchy for

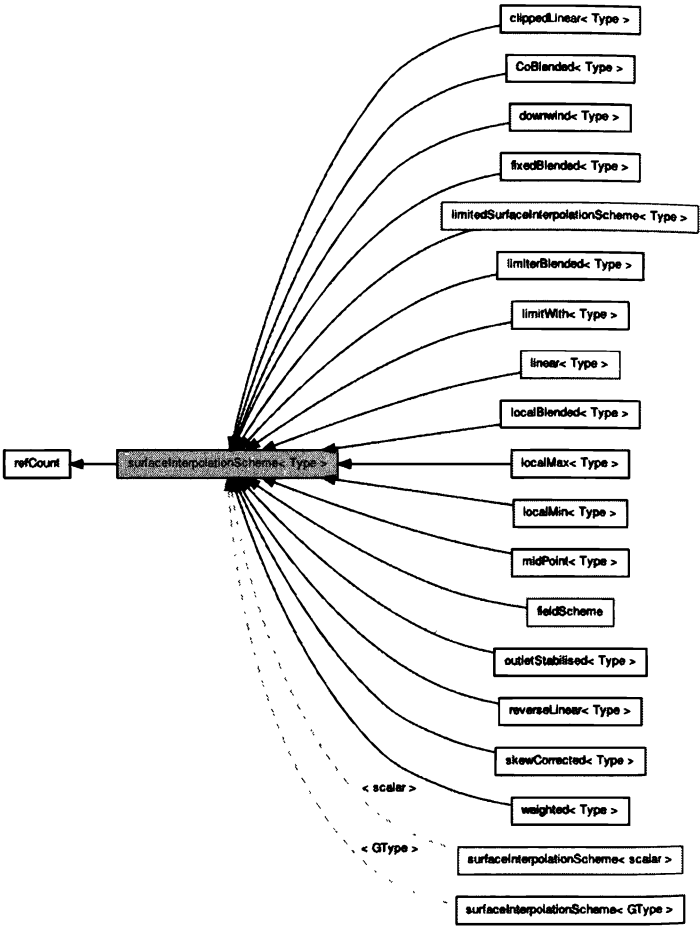


Figure 5.3: Hierarchy of the interpolation schemes based on owner-neighbour addressing.



the interpolation schemes has been built, as shown in Figure 5.3 with the `surfaceInterpolationScheme` at the root.

From the software design perspective, the interpolation schemes are encapsulated into classes that form a hierarchy. Some schemes share attributes or functionalities, so it makes sense to organize them as a class hierarchy. As a consequence of this kind of organization, the RTS mechanism allows the user to select different discretization/interpolation schemes at the start (or even during) a simulation. No re-compilation of the program code is required.

5.2.5 Post-processing

The post-processing activity can be performed after or during the simulation. When post-processing is done after the simulation has finished it will involve executing post-processing applications. OpenFOAM provides another distinct method to post-process the data during the simulation by invoking *function objects*.

Post-processing applications are distributed together with OpenFOAM or are written by the users themselves.

WARNING

Before programming a post-processing application, check to see if it already exists. OpenFOAM provides a large number of utility applications to choose from.

The `$FOAM_APP/utilities/postProcessing` directory holds all the post-processing applications which are distributed together with OpenFOAM, categorized into different groups.

The `$FOAM_APP/utilities/postProcessing` directory and its sub-directories are the first places to look for an existing post-processing application as shown in figure 5.4. If a new application is necessary it is likely that an application which closely fulfills the requirements of the user already exists somewhere in this directory. The post-processing applications are typically used to compute some integral quantity based on the fields stored during the simulation or to sample field values in

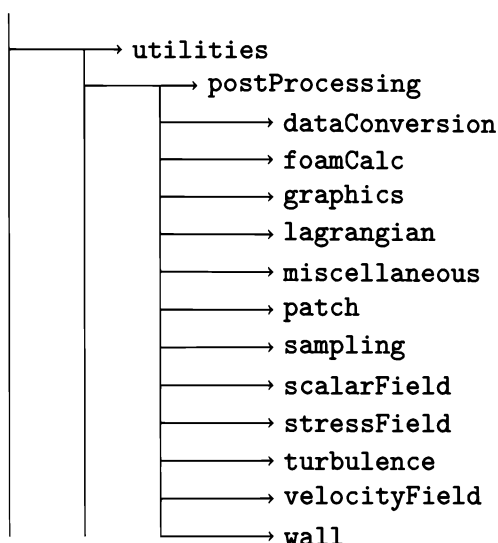


Figure 5.4: Application categories in OpenFOAM.

specific parts of the flow domain.

Function objects, unlike the post-processing applications, are called during the simulation run. The term *function object* comes from the C++ language terminology, where it denotes a class which is *callable*, since it implements a call operator - `operator()`. The function object is basically a function encapsulated into a class, which is advantageous e.g. when the function needs to store information about its state after execution.

For example, a function object which computes an average maximum pressure in a simulation can stop a simulation if the pressure value exceeds a prescribed value. Such a function needs access to the maximal pressure value and needs to store at least the number of times it has already been called in order to perform the averaging. Those two attributes are therefore encapsulated into a function object class. More information on function objects in OpenFOAM can be found in chapter 12.

5.3 Often Encountered Classes

In the previous section, as parts of OpenFOAM are discussed that the user comes into contact with when running simulations, some classes are only briefly mentioned. This section provides a deeper insight as well as best practices of some selected and frequently used classes in OpenFOAM.

5.3.1 Dictionary

The dictionary class is likely one of the first classes an OpenFOAM user interacts with. While the dictionary class interface is not terribly complex, some aspects of it may not be apparent a novice user.

Reading Data from a Dictionary

Reading dictionary entry values is the most basic form of working with the dictionary class. The dictionary class interface provides multiple methods that can be used to read data. The `lookupOrDefault` method is the commonly used example of this as it not only provides read access to the specified dictionary entry, it also defines a default value if no entry is found. This eliminates runtime errors caused by missing non-critical input values.

```
const dictionary& solution(mesh.time().solutionDict());
const word name(solution.lookupOrDefault<word>("parameter1"));
const vector vector1(solution.lookupOrDefault<vector>("vector1"));
```

As shown in the above code, `lookupOrDefault` requires more than just the name of the parameter to read. It also requires a template argument with the name of the data type that is to be looked up in the dictionary.

Accessing the Table of Contents

The table of contents contains the names of sub-dictionaries of a given dictionary. In the following example, the table of contents of the `fvSolution` dictionary file read by the `Foam::Time` class is accessed:

```
const dictionary& fvSolutionDict(mesh.time().solutionDict());
Info<< fvSolutionDict.toc() << endl;
```

An application example for the `toc()` method is the instantiation of an arbitrary number of objects of class `A` and storing them in a `List`. Each of the sub-dictionaries contains the required input parameters for that specific class. Consider the following very simplified class:

```
class A
{
    public:
        A(const dictionary& dict);
};
```

The objects of class `A` can be constructed multiple times so that the constructor arguments are stored as contents of a dictionary file `A`. In the following example, the file is saved in the `constant` directory of the simulation case.

```
// Construct the dictionary with a file name argument.
const dictionary dict(fileName("constant/A"));

// Get constant access to it's table of contents.
const wordList& toc(dict.toc());

// Initialize a list of objects of type 'exampleType'.
List<exampleType> myList(toc.size());

// Set the list of 'exampleType' objects equal to 'exampleType'
// objects initialized by the 'toc' sub-dictionary parameters.
forAll(myList, li)
{
    myList[li] = exampleType(dict[toc[li]]);
}
```

EXERCISE

What are the requirements prescribed to the type `'exampleType'` by the above example code?

Accessing Sub-dictionaries

Accessing sub-dictionaries is a task encountered very frequently while developing a user interface. Say the dictionary `A`, located in the `constant` directory, contains the following data:

```
axis    (1 0 0);
origin  (5 10 15);
type    "modelA";
```



```

modelA
{
    name    "uniform";
}

```

In order to access the `modelA` sub-dictionary in the simplest form, the following code suffices:

```

const dictionary dict(fileName("constant/A"));
const dictionary& subDict(dict.subDict("modelA"));

```

Providing a more useful implementation as the one above is always desirable. Rather than hard-coding the name of the sub-dictionary to be read, this can be done in a way that the user can decide during run-time:

```

const dictionary dict(fileName("constant/A"));
const word& type(dict.lookup("type"));
const dictionary& subDict(dict.subDict(type));

```

5.3.2 Dimensioned Types

Dimensioned types are used to attach units to properties. This is most often the case for scalar, vector and tensor types. They extend the tensor arithmetic operations to also include dimension checking capabilities. For example, both the velocity U and momentum ρU are vectors. Dimension checking in OpenFOAM is implemented by a class template `dimensioned<Type>`.

The `dimensioned<Type>` is a wrapper (adapter) class that delegates the computation of the tensor arithmetic to the wrapped tensor `Type` and the dimension checking to the wrapped `dimensionSet` object. Investigating the arithmetic operators of `dimensioned<Type>` class template leads to the implementation of the `+=` operator shown in listing 19. It can be seen that there are two arithmetic operations being performed - one for the dimension (units) and the other for the numerical value of the tensor. The arithmetic operators of the `dimensionSet` class are responsible for the dimension checking process. The source code of the `+=` arithmetic operator of the `dimensionSet` class is shown in listing 20. It provides enough information to conclude exactly how the dimension check is performed: a fatal error which aborts the program execution is generated when the

Listing 19 The += arithmetic operator of dimensioned<Type>.

```
template<class Type>
void Foam::dimensioned<Type>::operator+=
(
    const dimensioned<Type>& dt
)
{
    dimensions_ += dt.dimensions_;
    value_ += dt.value_;
}
```

Listing 20 The += arithmetic operator of dimensionSet.

```
bool Foam::dimensionSet::operator+=(const dimensionSet& ds) const
{
    if (dimensionSet::debug && *this != ds)
    {
        FatalErrorIn("dimensionSet::operator+=(
            const dimensionSet&) const")
            << "Different dimensions for += " << endl
            << "    dimensions : " << *this << " = "
            << ds << endl << abort(FatalError);
    }

    return true;
}
```

addition operation is performed for unequal (!=) dimensioned sets, provided dimension checking is turned on. The dimensionSet class implements dimensions as a set of integer exponents of the physical measures as shown in listing 21.

As shown in listing 22, the != dimension check operator is implemented in terms of the equality operator ==): The check procedure above loops over the dimensions of the operating dimensionSets. Through the loop it tests if the magnitude of the difference between dimension exponents is large enough to consider the sets unequal (> smallExponent). The value of smallExponent is a class-static variable:

```
atic const scalar smallExponent;
```

initialized to SMALL, which is equal to $1e - 15$ for double precision



Listing 21 Dimension exponents of dimensionSet.

```
//- Define an enumeration for the names of the dimension exponents
enum dimensionType
{
    MASS,           // kilogram   kg
    LENGTH,         // metre     m
    TIME,           // second    s
    TEMPERATURE,    // Kelvin     K
    MOLES,          // mole      mol
    CURRENT,        // Ampere    A
    LUMINOUS_INTENSITY // Candela   Cd
};
```

Listing 22 Operator == of the dimensionSet.

```
bool Foam::dimensionSet::operator==(const dimensionSet& ds) const
{
    for (int Dimension=0; Dimension < nDimensions; ++Dimension)
    {
        if
        (
            mag(exponents_[Dimension] - ds.exponents_[Dimension])
            > smallExponent
        )
        {
            return false;
        }
    }

    return true;
}
```

scalars.

Turning on dimension checking is done by setting the debug flag to "on" for the dimensionedSet class in the \$WM_PROJECT_DIR/etc/controlDict:

```
DebugSwitches
{
    Analytical          0;
    APIdiffCoefFunc     0;

    ...

    dictionary          0;
```

```
dimensionSet      1;  
mappedBase       0;  
  
...
```

The conclusions presented for the `+=` operator are exactly the same for other dimensioned tensor arithmetic operations. Note that dimension checking is activated by default.

EXERCISE

Why doesn't the arithmetic operator `+=` of the `dimensionSet` class perform an actual arithmetic operation when the `if` block is evaluated as `false`?

TIP

For many class templates in OpenFOAM there is a `typedef` (name synonyms) provided. In the C++ programming language the `typedef` keyword allows the programmer to define shorter and more concise type names. In the case of `dimensioned<Type>` the emphasis is not on name length, but on code style - `dimensionedVector` is a name in *camel case* which is used for types in OpenFOAM application level code.

The most popular alternative type names for `dimensioned<Type>` are `dimensionedScalar` and `dimensionedVector` and are used in the following examples. Also, note that the dimensioned types are constructed in a slightly more complex manner than what might be expected from what is described so far. Rather than requiring the tensor value and the dimensioned set, the dimensioned types require an additional parameter - a name. For example, if two `dimensionedVector` objects are constructed in the following way:

```
dimensionedVector velocity  
(  
    "velocity",  
    dimLength / dimTime,  
    vector(1,0,0)  
);  
  
dimensionedVector momentum  
(
```




```

    "velocity",
    dimMass * (dimLength / dimTime),
    vector(1,0,0)
);

```

Note that there are some pre-defined dimensionedSet objects used to initialize the velocity and momentum dimensionedVector objects: dimLength, dimTime and dimMass. Those are constant program-global objects, and they can be found in the source file dimensionedSets.C:

```

const dimensionSet dimless(0, 0, 0, 0, 0, 0, 0);

const dimensionSet dimMass(1, 0, 0, 0, 0, 0, 0);
const dimensionSet dimLength(0, 1, 0, 0, 0, 0, 0);
const dimensionSet dimTime(0, 0, 1, 0, 0, 0, 0);
const dimensionSet dimTemperature(0, 0, 0, 1, 0, 0, 0);
const dimensionSet dimMoles(0, 0, 0, 0, 1, 0, 0);
const dimensionSet dimCurrent(0, 0, 0, 0, 0, 1, 0);
const dimensionSet dimLuminousIntensity(0, 0, 0, 0, 0, 0, 1);

```

TIP

Because the fundamental physical dimension units are used to construct the complex ones (e.g. $N = \text{kgm/s}^2$), use the global predefined dimensionSet objects when defining your own dimension sets to improve code readability.

Moving on to arithmetics of dimensioned types and adding velocity to momentum like this:

```
momentum += velocity;
```

will result with a following error at the program execution:

```

--> FOAM FATAL ERROR:
Different dimensions for +=
dimensions : [1 1 -1 0 0 0 0] = [0 1 -1 0 0 0 0]

```

```

From function dimensionSet::operator+=(const dimensionSet&) const
in file dimensionSet/dimensionSet.C at line 179.

```

```
FOAM aborting
```

TIP

The dimension checking process in OpenFOAM is performed at run-time. As a result, code which contains errors in dimension operations will compile, but will not run. If dimension checking plays a significant role in your program (developing new mathematical models in OpenFOAM), test the program execution more often.

EXERCISE

Turn off dimension checking using the aforementioned debug flag and run the arithmetic addition for momentum and velocity.

5.3.3 Smart Pointers

Pointers are a special kind of variable which stores the memory address of an object and can be used to refer to that object effectively. When a pointer is passed to a function as an argument, or returned from it as result, those operations usually involve shorter execution times. In C++ it is possible to pass objects either by *value* or by *reference*, with the latter being significantly faster for larger objects. This is not only faster but also more converative in terms of memory usage as no data is copied. Returning larger objects by value and passing them as function arguments can be much more computationally expensive, compared to working with only their pointers. A good example for the application of pointers in OpenFOAM are the functions which implement interpolation algorithms. Interpolations operate on fields, which in CFD often have hundreds of thousands of components per CPU core. If the interpolation algorithm is implemented as a function, the function can provide the necessary field result in two ways, by returning the result as an object

```
result = function(input)
```

or by modifying a result argument that is passed as a non-constant reference to the function

```
function(result, input)
```



The first option is preferred for discretization algorithms (operators) because they are often composed of arithmetic expressions to build mathematical models. For example, consider the momentum conservation equation code taken from the `interFoam` solver:

```
fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
  + fvm::div(rhoPhi, U)
  + turbulence->divDevRhoReff(rho, U)
);
```

The result of the sum of operators acting upon fields `rho`, `U` and `rhoPhi` will be a coefficient matrix (`fvVectorMatrix`). As a consequence, the following points must be fulfilled from the above code:

- `fvVectorMatrix` needs to be copy-constructable,
- all operators must *return* a `fvVectorMatrix`,
- the `fvVectorMatrix` addition operator must return a `fvVectorMatrix`.

If the operators `ddt` and `div` had been implemented as functions that take modifiable parameters, writing mathematical models easily (usually referred to as *equation mimicking*) would not be possible. The matrices that are returned by the functions are quite large, so returning them by value introduces the penalty of creating temporary objects. OpenFOAM avoids creating the unnecessary copy operations in an explicit way not relying on compiler optimizations, with the help of natively implemented OpenFOAM *smart pointers*. A smart pointer is initialized within the interpolation function, and is returned by value. The same goes for other operations involving equation discretization, such as convection schemes.

For example, the `fvmDiv` divergence operator of the Gauss convection scheme initializes such a smart pointer (`tmp<fvMatrix<Type>>`) is shown in listing 23. Afterwards it performs the calculations that are responsible for defining the elements in the coefficients matrix shown in listing 24. The *smart pointer* initialized at the beginning of the function is then returned by value:

```
return tfvm;
```

Listing 23 Smart pointer used by the Gauss convection scheme div operator.

```
tmp<fvMatrix<Type> > tfvm
(
    new fvMatrix<Type>
    (
        vf,
        faceFlux.dimensions()*vf.dimensions()
    )
);
```

Returning a smart pointer to the finite volume matrix (tfvm) by value results in a copy operation *of the smart pointer*. However, there is a significant difference between a copy of the smart pointer and a copy of an entire matrix object: the pointer's value is only the address of the matrix, and not the entire matrix itself. This approach substantially increases the efficiency of the implementation.

TIP

Avoiding unnecessary copy operations has a commonly used shorter name: *copy elision*. Copy elision can be enforced in various ways in the C++ programming language: compiler optimizations (Return Value Optimization, Named RVO), Expression Templates (ET), or by using rvalue references and move semantics provided by the C++11 language standard.

In general, when pointers are used, they point to objects which are created on the heap using the operator `new`:

```
someType* ptr = new someType(arguments...);
```

Since the C++ programming language does not, on purpose, support automatic garbage collection¹, the programmer is left responsible for releasing the resources.

Thus, each invocation of the `new` operator needs to be followed by a corresponding call to an appropriate `delete` operator. Of course, the

¹Automatic garbage collection refers to the deletion of heap-allocated objects.



Listing 24 Matrix coefficient calculation by the Gauss convection scheme.

```

fvm.lower() = -weights.internalField()*faceFlux.internalField();
fvm.upper() = fvm.lower() + faceFlux.internalField();
fvm.negSumDiag();

forAll(vf.boundaryField(), patchI)
{
    const fvPatchField<Type>& psf =
        vf.boundaryField()[patchI];
    const fvsPatchScalarField& patchFlux =
        faceFlux.boundaryField()[patchI];
    const fvsPatchScalarField& pw =
        weights.boundaryField()[patchI];

    fvm.internalCoeffs()[patchI] =
        patchFlux*psf.valueInternalCoeffs(pw);
    fvm.boundaryCoeffs()[patchI] =
        -patchFlux*psf.valueBoundaryCoeffs(pw);
}

if (tinterpScheme_().corrected())
{
    fvm += fvc::surfaceIntegrate
        (
            faceFlux*tinterpScheme_().correction(vf)
        );
}

```

call to the `delete` operator must be placed in an appropriate location, such as a class destructor. This opens the possibility that the programmer simply forgets to delete pointers, which results in memory leaks. As an alternative source of errors, accessing a part of the memory which is referred to by an already deleted pointer will lead to undefined behavior. Both issues will occur at runtime and are usually sources of errors which are notoriously difficult to find and debug. To circumvent both problems, direct handling of raw pointers is to be avoided. In C++, handling of raw pointers has been replaced by an idiom called Resource Acquisition Is Initialization (RAII).

The RAII idiom states that raw pointers need to be encapsulated into a class, whose destructor takes care of deleting the pointer and releasing the resource at the appropriate place in the code. Adapting raw pointers in such classes, and providing different functionalities to the adapted raw pointers has lead to the development of so-called *smart pointers*. Differ-

ent smart pointers exist and provide different functionalities. OpenFOAM implements and two of such smart pointers: `autoPtr` and `tmp`.

Provided that the environmental variables are set by sourcing the `etc/bashrc` configuration script of the example code repository, the example application code is made available in the folder `$PRIMER_EXAMPLES_SRC/applications/test/`. Those readers that are interested in following the tutorials presented in the next sections step-by-step, need to create a new executable application - `testSmartPointers`. Creating new applications in OpenFOAM is simplified, since scripts are available that generate *skeleton* directories for applications. To create a new application, a directory is chosen where the application code will be placed, and the following commands are executed:

```
mkdir testSmartPointers
cd testSmartPointers
foamNew source App testSmartPointers
sed -i 's/FOAM_APPBIN/FOAM_USER_APPBIN' Make/options
```

The final line replaces the placement directory for the application binary file from the platform directory, to the user application binaries directory.

TIP

It is a good practice to build your own applications into the `$FOAM_USER_APPBIN`, which needs to be specified in the `Make/options` build configuration file.

WARNING

In this section, the `gcc` compiler is used. Be aware of this when reading about specific compiler flags in the text.

Using the `autoPtr` Smart Pointer

In order to illustrate how the `autoPtr` is used, a new class needs to be defined for the used examples. This class should notify the user each time its constructors and destructor are called. For sake of this example, it inherits from the basic field class template `Field<Type>`, and is named



`infoField`. Any other class could be used, since the optimizations performed by the compiler for copy operations do not depend on the size of the objects.

Listing 25 Class template `infoField`.

```
template<typename Type>
class infoField
:
    public Field<Type>
{
public:
    infoField()
    :
        Field<Type>()
    {
        Info << "empty constructor" << endl;
    }

    infoField(const infoField& other)
    :
        Field<Type>(other)
    {
        Info << "copy constructor" << endl;
    }

    infoField (int size, Type value)
    :
        Field<Type>(size, value)
    {
        Info << "size, value constructor" << endl;
    }

    ~infoField()
    {
        Info << "destructor" << endl;
    }

    void operator=(const infoField& other)
    {
        if (this != &other)
        {
            Field<Type>::operator=(other);
            Info << "assignment operator" << endl;
        }
    }
};
```

To start, the `infoField` class template can be defined as shown in listing 25. The class template inherits from `Field<Type>` and uses the

following:

- empty constructor
- copy constructor
- destructor
- assignment operator

Each time those functions are used, an Info statement signals the particular call to the standard output stream. This is solely done for the purpose of obtaining information on which function has been executed.

To continue with the example, a function template needs to be defined which returns an object by value:

```
template<typename Type>
Type valueReturn(Type const & t)
{
    // One copy construction for the temporary.
    Type temp = t;

    // ... operations (e.g. interpolation) on the temporary variable.

    return temp;
}
```

The name of the type used in the example is shortened in order to reduce the amount of unnecessary typing:

```
// Shorten the type name.
typedef infoField<scalar> infoScalarField;
```

In the main function the following lines are implemented:

```
Info << "Value construction : ";
infoScalarField valueConstructed(1e07, 5);

Info << "Empty construction : ";
infoScalarField assignedTo;

Info << "Function call" << endl;
assignedTo = valueReturn(valueConstructed);
Info << "Function exit" << endl;
```

Compiling and executing the application with either Debug or Opt options will produce exactly the same results, even though the Debug option turns



of compiler optimizations. As mentioned at the section beginning: the compiler is very clever at recognizing the fact that a temporary object is returned only to be discarded away after assignment. Another advantage of not using the geometrical fields is that this small example application does not need to be executed within an OpenFOAM simulation case directory. It can be called directly from the directory where the code is stored. Executing the application results in the following output:

```
?> testSmartPointers
```

```
Value construction : size, value constructor
Empty construction : empty constructor
Function call
copy constructor
assignment operator
destructor
Function exit
destructor
destructor
```

Considering each line of the output individually, while comparing it to the `valueReturn` function code, interestingly reveals that *the temporary return variable was never constructed*. A copy construction is missing for the `return` statement of the function, since the function *returns by value*, as well as a corresponding destructor call. The reason for this kind of behavior is the *copy elision* optimization performed automatically by the compiler, that is present even in the Debug mode. In order to remove this optimization, an additional compiler flag can be added to the `Make/options`:

```
EXE_INC = \
  -I$(LIB_SRC)/finiteVolume/lnInclude \
  -fno-elide-constructors

EXE_LIBS = \
  -lfiniteVolume
```

The `gcc` compiler flag `-fno-elide-constructors` will prevent the compiler from performing optimizations, important for the functions such as `valueReturn`.

Compiling and running the application with the options file as defined



WARNING

Remember to call `wclean` before executing `wmake`. Modifications to the `Make/options` file are not recognized by the `wmake` build system as a source code modification that requires re-compilation.

above results in the following output²:

```
?> testSmartPointers
Value construction : size, value constructor
Empty construction : empty constructor
Function call
copy constructor # Copy construct tmp
copy constructor # Copy construct the temporary object
destructor # Destruct the tmp - exiting function scope
assignment operator # Assign the temporary object to assignedTo
destructor # Destruct the temporary object
Function exit
destructor
destructor
```

The output shows the unnecessary creation and deletion of the temporary object. Eliding copies of temporary objects by performing an in-place construction of an object at the point where the function returns has become a standard option for compilers. It happens regularly that this feature cannot be disabled even when surpressing optimizations in Debug mode with the compiler flags:

```
-O0 -DFULLDEBUG
```

To disable the constructor copy eliding optimization the compiler flag `-fno-elide-constructors` had to be passed explicitly in `Make/options`.

The above paragraphs dealt with when and how `autoPtr` should not be used, whereas some proper applicatons of the `autoPtr` are discussed in the following.

Some of the most prominent examples of the valid usage of `autoPtr` are within models employing RTS. The particular classes are instantiated

²Please note that the comments, starting with a # are added by the authors



TIP

The above example serves one major purpose: To show that it is not necessary to use the `autoPtr` smart pointer in expressions where a named temporary object is returned anyway. On modern compilers, even when compiling in the debug mode (for OpenFOAM this means setting `$WM_COMPILE_OPTION` to Debug), this optimization is turned on by default.

due to the keywords specified in a dictionary. Those models are e.g. the turbulence models in the solver application code. RTS is covered in great detail but in short: it allows class users to instantiate objects of a specific class in a class hierarchy at runtime. Instantiating objects in that way enables the ability of C++ to access the derived class object via a base class pointer or reference. This is usually referred to as *dynamic polymorphism*.

WARNING

It is impossible to cover all details of the C++ language standard used by OpenFOAM in this book. Whenever a C++ construct is encountered that does not sound familiar, it should be looked up on the internet.

The turbulence models are typically instantiated in the particular solver's `createFields.H`, which is included before the beginning of the time loop. The relevant lines are the following:

```
autoPtr<incompressible::RASModel> turbulence
(
    incompressible::RASModel::New(U, phi, laminarTransport)
);
```

Obviously, an `autoPtr` is used to store the turbulence model. If raw pointers would have been used to access the `RASModel` object, another line in the solver code would be required which deletes the raw pointer. With `autoPtr` being a smart pointer, this deletion is performed within the `autoPtr` destructor. The code for the raw pointer method would look something like this:

```
incompressible::RASModel* turbulence =
```

```
incompressible::RASModel::New(U, phi, laminarTransport)
```

At the end of the solver code, the following lines would be required:

```
delete turbulence;  
turbulence = NULL;
```

Of course, for a single pointer, adding the lines to release resources might not be a problem. However, the RTS is used for: transport models, boundary conditions, discretization schemes, interpolation schemes, gradient schemes and so forth. All of those objects are small compared to things like fields and the mesh, so could they be returned by value? From the standpoint of efficiency - yes, especially considering the Return Value Optimization (RVO) is implemented by all modern compilers. From the standpoint of flexibility - there is no chance of doing that, since the dynamic polymorphism relies on access via pointers or references. Keeping the runtime flexibility high, means relying on pointers or references.

TIP

Using `autoPtr` or `tmp` is necessary when RTS is used to select objects at runtime.

The following example examines the `autoPtr` interface and its problematic ownership based copy semantics. The `autoPtr` owns the object it points to. This is expected, as RAII requires the smart pointer to handle the resource release so the programmer doesn't have to. As a consequence, creating copies of `autoPtr` is complicated - copying an `autoPtr` invalidates the original `autoPtr` and transfers the object ownership to the copy. To see how this works, consider the main function of the following code snippet:

```
int main()  
{  
    // Construct the infoField pointer  
    autoPtr<infoScalarField> ifPtr (new infoScalarField(1e06, 0));  
  
    // Output the pointer data by accessing the reference -  
    // using the operator T const & autoPtr<T>::operator()  
    Info << ifPtr() << endl;  
  
    // Create a copy of the ifPtr and transfer the object ownership
```



```

// to ifPtrCopy.
autoPtr<infoScalarField> ifPtrCopy (ifPtr);

Info << ifPtrCopy() << endl;

// Segmentation fault - accessing a deleted pointer.
Info << ifPtr() << endl;

return 0;
}

```

It is important to notice that once a copy of the `autoPtr` object is performed, the ownership of the object pointed to is transferred. Commenting out the line that causes a segmentation fault, the resulting application output looks like this:

```

?> testSmartPointers
size, value constructor
1000000{0}
1000000{0}
destructor

```

Obviously only a single constructor and destructor of the `infoScalarField` class are invoked, even though the `autoPtr` objects are passed by value. Hence `autoPtr` can be used to save unnecessary copy operations.

Using the `tmp` Smart Pointer

The `tmp` smart pointer prevents unnecessary copies of objects by performing reference counting. Reference counting is a process where the same object is being passed around. It is wrapped by the smart pointer, with the number of references to this object being increased each time a copy or assignment of the smart pointer is performed.

Complying with RAII, the destructor of the `tmp` pointer is responsible for destroying the wrapped object. The destructor checks the number of references the object has in the current scope. If this number is greater than zero, the destructor simply reduces it by one and allows the object to live on. As soon as the destructor is called in a situation where the reference count to the wrapped object has reached zero, the destructor deletes the object.

There is a catch when using the `tmp` smart pointer: the pointer class

TIP

The definition of the tmp smart pointer can be found in \$FOAM_SRC/OpenFOAM/memory/tmp.

template is not made responsible for counting the references. Reference counting is expected from the wrapped type. This can be easily checked when examining the class destructor shown in listing 26.

Listing 26 Destructor of the tmp class template.

```
template<class T>
inline Foam::tmp<T>::~tmp()
{
    if (isTmp_ && ptr_)
    {
        if (ptr_->okToDelete())
        {
            delete ptr_;
            ptr_ = 0;
        }
        else
        {
            ptr_->operator--();
        }
    }
}
```

The ptr_ variable is the wrapped raw pointer to an object of type T and the destructor attempts to access two member functions:

- T::okToDelete()
- T::operator--()

As a result, the wrapped object must adhere to a specific interface. Another way to test this catch is to try and use the tmp with a trivial class that can be define as follows:

```
class testClass {};
```

Then trying to wrap this class into a tmp smart pointer:



```
tmp<testClass> t1(new testClass());
```

The above code results in the following errors:

```
tmpI.H:108:9: error: 'class testClass'
has no member named 'okToDelete' if (ptr_>okToDelete())
```

```
tmpI.H:115:13: error: 'class testClass'
has no member named 'operator--' ptr_>operator--();
```

By means of this error, the compiler complains about the fact that the aforementioned member functions are not implemented by `testClass`.

Since OpenFOAM makes the objects perform the reference counting, it has been encapsulated into a class that such objects inherit from, named `refCount`. The `refCount` class implements the reference counter and the related member functions.

TIP

In order to use `tmp<class T>` in OpenFOAM, the type `T` of the wrapped object should inherit from `refCount`.

If the `testClass` is modified to inherit from `refCount`:

```
class testClass : public refCount {};
```

It can then be wrapped with `tmp`. To see how reference counting works and how unnecessary construction of objects is avoided, the `tmp` can be used with the class `infoScalarField`, that was used in the examples describing `autoPtr`. The `refCount` class allows the user to get information on the current reference count with the member function `refCount::count()`, which is used in the following example. Artificial scopes are used to decrease the life-span of `tmp` objects so that their destructors are called. This would occur in a regular program code when nested function calls or loops are present. Here is the example code:

```
tmp<infoScalarField> t1(new infoScalarField(1e06, 0));
Info << "reference count = " << t1->count() << endl;
{
    tmp<infoScalarField> t2 (t1);
```

```
Info << "reference count = " << t1->count() << endl;
{
    tmp<infoScalarField> t3(t2);
    Info << "reference count = " << t1->count() << endl;
} // t3 destructor called

Info << "reference count = " << t1->count() << endl;
} // t2 destructor called
Info << "reference count = " << t1->count() << endl;
```

This results in the following command line output:

```
>? testSmartPointers
size, value constructor
reference count = 0
reference count = 1
reference count = 2
reference count = 1
reference count = 0
destructor
```

A single constructor and the corresponding destructor output coming from the `infoField` class shows that only a single construction and destruction was performed, even though the `tmp` objects were passed by value as function arguments.

5.3.4 Volume Fields

In this section a brief description of the class interface is provided for volume fields. An indepth discussion of boundary fields and boundary conditions, including the theory behind them, is covered by chapter 10.

Volume fields are those fields typically used to store the field values mapped to the cell centres. Depending on the property stored by the field, either a `volScalarField`, `volVectorField` or a `volTensorField` can be used.

There are surface- and point- based fields as well, which store field values in the face centres and points, respectively. Note that all mentioned fields are constructed similarly, independent of their type.

The reason behind the similarity in field class interfaces is in the fact



that fields that map values to meshes in OpenFOAM are implemented by the `GeometricField` class template. The specific fields, such as volume fields, surface fields and similar other fields are then generated in the form of concrete classes by instantiating the `GeometricField` class template with specific template arguments. As noted in the section on the dimensioned types, the type names used for fields in OpenFOAM are shortened for convenience using the `typedef` keyword.

TIP

When problems are encountered in compiling code that uses field classes, such as `volScalarField`, template errors will occur. Because the fields are instantiations of the `GeometricField` template, the `GeometricField` class template source code is to be investigated.

Before being able to work with a field object, it must be constructed. The class interface has several constructors which may come in handy depending on the situation. The simple constructor is the *copy constructor*:

```
// Assuming that the volScalarField p exists
const volScalarField pOld(p);
```

As the name suggests, it constructs a copy of the original, which is type-identical. However, in order to use this constructor, a `volScalarField` must be present in the first place. Two approaches can be used for this: either read field data from a file or generate it from scratch. The initialization of a field based on an input file is fairly straightforward and can be found in nearly every solver of the OpenFOAM package. For that purpose, the `IObject` is leveraged for accessing that file and constructing a `volScalarField T`:

```
volScalarField T
(
    IObject
    (
        "T",
        runTime.timeName(),
        mesh,
        IObject::MUST_READ,
        IObject::AUTO_WRITE
    ),

```

```
    mesh  
);
```

The above code initializes the `volScalarField` based on the contents of the T file, which must be present. If the T file does not exist, the execution of the code will stop due to the `IObject::MUST_READ` instruction. Other instructions can be selected as well, depending particular needs.

In some circumstances, fields need to be constructed without reading data from a file. The flux field is a good example for that:

```
surfaceScalarField phi  
(  
    IObject  
    (  
        "phi",  
        runTime.timeName(),  
        mesh,  
        IObject::READ_IF_PRESENT,  
        IObject::AUTO_WRITE  
    ),  
    linearInterpolate(rho*U) & mesh.Sf()  
);
```

Here the `phi` field is constructed from the field data itself if the file is present. Otherwise, the flux is calculated from the velocity field directly.

Accessing Specific Cells

Specific cells can be addressed by invoking the `[]` operator on the particular field and passing the desired cell label as an argument. When performing mathematical operations with any field, it is not advisable to do this on a cell-to-cell basis. Instead, the operators of the fields should be used. In conclusion, this selection of specific cells should only be used

TIP

Using loops for fields in the application-level program code reduces code readability and may cause a significant drop in computational efficiency.



if a subset of cells needs to be used for the computation. The following code shows an example of how cells of the pressure and velocity fields can be selected:

```
labelList cells(3);
cells[0] = 1;
cells[1] = 42;
cells[3] = 39220;

forAll(cells, cI)
{
    Info<< U[cells[cI]] << tab << p[cells[cI]] << endl;
}
```

Accessing Boundary Fields

As covered in chapter 1, the internal field values are separated from the boundary field values. Such logical separation of cell centered and boundary (face-centered) values is defined by the principles of the numerical interpolations that support the FVM.

Separating the boundary fields has an important impact onto the way the algorithms are parallelized in OpenFOAM. Numerical operations are parallelized in OpenFOAM using data parallelism where the domain is decomposed into sub-domains and the numerical operations are executed on each separate sub-domain. As a result, a number of processes are executed which require communication with each other across the decomposition (processor) boundaries. Modeling the process boundary as a boundary condition results in automatic parallelization of all the numeric operations based on owner-neighbor addressing in OpenFOAM. Automatic parallelization of the code is a noteworthy feature of the entire OpenFOAM platform.

The following example shows how to access the boundary field values of a volumetric scalar field `pressure`, for the boundary patch `outlet`. The boundary field on the outlet can be found based on the ID mapped to the name of the boundary mesh patch (a sub-set of the boundary mesh):

```
label outletID = mesh.boundaryMesh().findPatchID("outlet");
```

and the boundary field is then accessed using the `GeometricField::boundaryField` member function:

```
const scalarField& outletPressure =  
    pressure.boundaryField()[outletID];
```

The volumetric field `pressure` has a member function `boundaryField` which returns a list of pointers to the boundary fields. The position of the outlet boundary field in that pointer list is defined by the `outletID` label (index). The above code snippet sets the boundary field as a constant reference `outletPressure`, however non-constant access to the boundary field is provided by the member function. This can be confirmed by observing the declaration of the `GeometricField` class template:

```
// - Return reference to GeometricBoundaryField  
GeometricBoundaryField& boundaryField();
```

The `GeometricBoundaryField` is a class template which is parameterized with the same parameters as the `GeometricField` and the definition of this class template is placed in the public part of the `GeometricField` class interface.

EXERCISE

Use a non-constant reference to a boundary field to force the modification of a `fixedValue` boundary field. Hints: This exercise requires understanding of programming a new OpenFOAM application as well as boundary conditions. Both topics are covered in chapters 8 and 10.

Further reading

Jasak, Hrvoje, Aleksandar Jemcov, and Željko Tuković (2007). “Open-FOAM: A C++ Library for Complex Physics Simulations”. In:



6

Productive Programming with OpenFOAM

Any software project of a significant size, whether it be independent or collaborative, demands some level of organization. From the directory organization to the Version Control System (VCS), there are many aspects of code development that call for systems and standards to be in place. These organizational constructs can facilitate improved productivity while, at the same time, make code easier to share and maintain.

Development of CFD applications puts a strong emphasis on computational efficiency because of the large datasets as well as the large number of computational operations involved in a numerical simulation. Without careful consideration of even simple algorithms, crippling bottlenecks can be created. Resolving problems in the code such as execution errors (bugs) or computational bottlenecks can typically be completed faster and easier with the appropriate tools.

Running simulations in parallel mode on an High Performance Computing (HPC) cluster requires the user to first install OpenFOAM on a cluster. Even in the case when a complete OpenFOAM installation is available on the cluster, having background knowledge on that topic makes it easier for the user to more accurately assess possible installation problems and report them to the cluster administrator.

In this chapter, various organizational and analytical practices are reviewed in hopes that they help increase productivity when programming within the OpenFOAM framework as well as installing OpenFOAM on an HPC cluster.

6.1 Code Organization

This section describes the organization of OpenFOAM sources and the workflow associated with it. It will often relate to the example code repository provided alongside with the book. Browsing through the appropriate repository code while reading the following text should aid in the understanding of the covered topics. When performing code development, the code itself should be organized into two layers: library code and application code. The library code may contain a single or multiple *libraries* that implement various re-usable computations. Sometimes the library level code is referred to as application logic. The library code is, by design, not specific to one single application, this code layer is often re-used by many executable applications. A library will contain declarations of functions or classes, as well as their implementation. The library is usually compiled into what is called 'object code', to save compilation times. The compiled library code is then linked to the application code by a linker program. Note that while libraries are indeed compiled code, they cannot be executed in the command line like executable application programs.

The application code on the other hand is using the library code to assemble a higher level functionality. A numerical solver is a good example of this in that it may combine separate libraries to handle conceptually distinct tasks such as disk I/O, mesh analysis, and matrix assembly.

When source code is organized into application and library layers, it is typically more easily extensible and can be shared more readily with others. Because OpenFOAM follows this structure the top-level directory `$WM_PROJECT_DIR` holds two sub-directories: `applications` and `src`. The `src` folder stores the various libraries while the `applications` folder stores the executable applications that make use of those libraries.

During the code development process for OpenFOAM, the programmer



faces two main possibilities for the code organization: programming within the OpenFOAM file structure, or programming within a separate file structure. Programming within the main library makes sense at first glance as it is logical to keep similar code in close proximity within the directory trees. This 'main file structure' approach to development can, unfortunately, quickly cause problems when collaborating and sharing version control systems. Version control systems are almost unanimously considered a must for anyone who is involved in a collaborative programming activity. Also, for personal use it can be of great benefit because of the speedup it brings in the development process. Even if a version control system is used properly, sharing custom code located within the main OpenFOAM repository with others might be problematic for the following reasons:

- there is not a clear overview of the project files,
- tutorials and test cases are separated from the accompanying code,
- placing accompanying libraries not distributed with OpenFOAM might be a problem,
- collaborating with others requires having access to a full-fledged OpenFOAM release.

The last point makes collaborative work difficult, as creating a clone of the entire release repository makes it necessary for anyone to clone the entire OpenFOAM platform in order to collaborate on a usually significantly smaller custom project. Those items could be addressed simply by integrating own work with an established community of programmers that are working with OpenFOAM. There are two communities nowadays that are revolving around either the official OpenFOAM release, or the foam-extend project. However, there are also cases when the projects are not to be shared with the general public from the point of their inception. Also, it is often beneficial to first investigate the feasibility (accuracy, efficiency, predicted development time) of the own project before integrating the development with an OpenFOAM release.

In such cases when the project is not to be integrated with an OpenFOAM release, bundling a library and application code in a single separate repository, that can be compiled directly, makes individual and collaborative development much easier. At a latter point, when the project proved to be of higher quality, integration into one of the OpenFOAM release

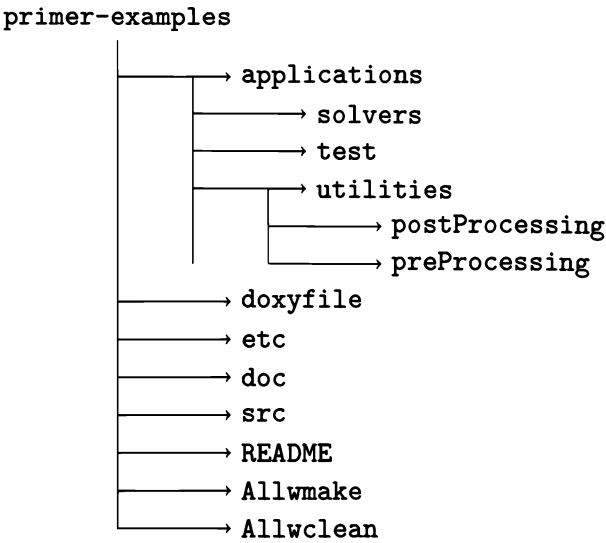


Figure 6.1: Directory structure of the example code repository.

projects can be performed. Nowadays there are multiple VCS hosting services supporting VCSs. These services provide advanced web interfaces allowing the user to use bug tracking, hosting a wiki for each project, and other tools that make working on such a project much easier. Two of the most popular ones are github and bitbucket, with the difference that bitbucket supports an unlimited number of free private repositories. This is great if the code should be shared with selected group of collaborators, without making it at first available to the general public.

6.1.1 Directory Organization

There are significant benefits in applying the same directory organization structure that is applied to the OpenFOAM platform to a custom OpenFOAM project. This includes simplified integration of that code within the platform, easier collaborative work. If the code is organized this way, the structure of the directories will be more self-explanatory and intuitive, especially to programmers already familiar with the directory structure of OpenFOAM. Maintaining a uniform directory structure is also a fundamental way of indirectly documenting the code.



To examine an example directory organization, consider the example code repository for the book and how it is organized.

As shown in figure 6.1, the applications directory is where the applications are stored. Inside the applications directory, similar to the OpenFOAM organization, the following subdirectories are present: `solvers`, `test` and `utilities`. The `etc` directory is used to configure the automatic compilation of the code. The organization of the `src` folder typically depends on the purpose of the library, but it should nonetheless be sensible. As the classes extract and encapsulate common behavior between different abstractions, so does layered code organization separate different collections of class implementations into separated linkable libraries. Organizing and separating library categories reduces the size of the compiled application code and makes the compilation process faster.

The README file usually found in the top directory of the code repository is useful because it is the first file that is read when a user starts working with new code. A general description of the background of the project and its most important applications, as well as the up-to-date links to external documentation sources and forums, can usually be found there. Local documentation of the code can be generated using the Doxygen documentation system, which uses the `doxyfile` to specify the files as well as the details involving the look and feel of the generated HTML documentation.

6.1.2 Automating Installation

If the code is organized in a self-sustained structure, it is usually expected that the package will facilitate an simplified, automated build process. OpenFOAM uses its own build system called `wmake`, which makes use of various environmental variables to automatize the compilation and linking of the library and application code. The same approach can be applied to a custom code repository and is implemented for the provided example code repository. Listing 27 shows the simple `bashrc` configuration script for the example code repository. It configures the path variable named `$PRIMER_EXAMPLES`, which is the path variable to the main folder of the code repository. The `PATH` variable needs to be updated, since the example code repository contains scripts in OpenFOAM, that are located in `src/scripts`. Otherwise those scripts cannot be invoked from

Listing 27 The `etc/bashrc` configuration script for the example code repository.

```
#!/bin/sh

DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

export PRIMER_EXAMPLES=${DIR%/etc*}

export PRIMER_EXAMPLES_SRC=${PRIMER_EXAMPLES/src}

export PATH=${PRIMER_EXAMPLES_SRC/scripts}:${PATH}
```

anywhere in the filesystem. This is another point for maintaining the directory structure of the repository similar to the one of OpenFOAM: a skeleton repository directory structure can be stored with the configuration script as well as the compilation scripts. It can then be simply reused as another repository by changing the pathname variable of the repository. The applications and libraries of the code repository rely on the variable `$PRIMER_EXAMPLES` to find the directories that hold the header files that are to be included before compilation.

The usual compilation scripts, `Allwmake` and `Allwclean`, are present in the base directory as well as in the `src` and `applications` sub-directories. An example content of the `Allwmake` script for building libraries within the `src` directory is shown in listing 28. The `wmakeLn`-

Listing 28 The `Allwmake` build script for library code.

```
#!/bin/sh
cd ${0%/*} || exit 1    # run from this directory

wmakeLnInclude -f .
wmake libso exampleLibrary
```

`LnInclude` script searches recursively through the current directory, finds all OpenFOAM source files, and create symbolic links to these files in the `src/LnInclude` directory. This greatly simplifies the configuration of the building process. As soon as the absolute path of the repository folder is defined (`$PRIMER_EXAMPLES` variable defined by `etc/bashrc` script), the inclusion of header files holding the class declarations is relying on the symbolic links of all source files stored in the `src/LnInclude` directory. The name of library to be compiled (`exampleLibrary`) is passed



as an argument to the `wmake` tool together with the `libso` option which ensures the building process will result in a dynamically linkable library.

WARNING

In the versions above and including 2.2.x of the official OpenFOAM release, the `libso` option is no longer necessary for the library compilation. It is shown here, however, in the event that the example code is compiled against an older release version.

Application code of OpenFOAM is typically stored in a directory named the same way as the application. The contents of an example application directory are shown in figure 6.2. The `applicationName.C` is the source file of the application. Both, files and options files are used by the `wmake` build system to compile the application code. Examples of both

Listing 29 The Make/files file.

```
applicationName.C
```

```
EXE = $(FOAM_USER_APPBIN)/applicationName
```

files and options files are listed in listing 29 and 30, respectively. The application installation target directory is set to `$FOAM_USER_APPBIN`, in order not to pollute the primary OpenFOAM system application directory with custom applications. The options file shows that the custom application `applicationName` relies on the repository variable `$PRIMER_EXAMPLES_SRC` and the OpenFOAM generated `lnInclude` directory, to locate the required header files. Additionally, the application will link to the library `exampleLibrary` and use the needed functional-

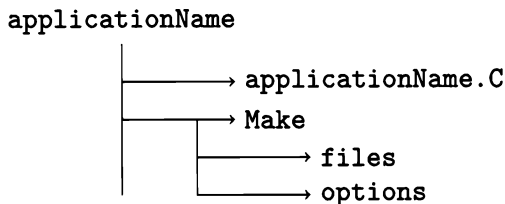


Figure 6.2: Application directory structure.

Listing 30 The Make/options file

```
EXE_INC = \  
-I$(LIB_SRC)/finiteVolume/lnInclude \  
-I$(PRIMER_EXAMPLES_SRC)/lnInclude  
  
EXE_LIBS = \  
-L$(FOAM_USER_LIBBIN) \  
-lfiniteVolume \  
-lexampleLibrary
```

ity contained within it.

TIP

The key step in having a custom project directory structure is preparing the `bashrc` configuration script. Relying on the variables set by that script to locate header files and libraries of the custom project separates the custom project from the OpenFOAM platform.

TIP

Note that the libraries of the example repository are also compiled to the user directory, to avoid polluting system OpenFOAM directories. Custom code should never be compiled into `$FOAM_APPBIN/$FOAM_LIBBIN`.

This kind of configuration is a simple way of working and programming with custom code. The following list is a summary of steps in the workflow described above.

- source `./etc/bashrc` to set the `$PRIMER_EXAMPLES` variable for the shell terminal,
- add `source /path/to/code/directory/etc/bashrc` to the startup script of the shell *if you want to permanently set the environment variable*,
- execute `./Allwmake` in the top code directory to compile all libraries and applications,
- execute `./Allwclean` in the top code directory for clean up of binaries,
- when you add a library e.g. `libraryName` to the repository, edit



`src/Allwmake` and add `wmake libso libraryName` for compilation, as well as `wclean libraryName` in `src/Allwclean` for cleanup of binaries for the new library code,

- if you use standard directories in the example code repository for your applications, they will be compiled/cleaned without editing the compilation/cleanup scripts.

As the example code repository will evolve with time, the up-to-date description on how to obtain the code, configure the repository and compile its libraries and applications can be found in the distributed **README** file.

6.1.3 Documenting Code using Doxygen

Doxygen is an automatic documentation generation system that is able to create a set of interlinked HTML files describing the classes, their attributes, member functions as well as inheritance and collaboration diagrams. More details on Doxygen can be obtained from the project website www.stack.nl/~dimitri/doxygen/.

The example code repository has a doxygen configuration file (`doxyfile`), which is set to search the repository directory structure and find OpenFOAM source files. The documentation of the classes is using UML, as it provides more detail than the standard Doxygen graphical class description.

To generate the documentation, execute the `doxygen` command using `doxyfile` as its argument in the main folder of the example code repository:

```
?> doxygen doxyfile
```

The documentation will be generated and saved in the `doc` directory and can be viewed by pointing any browser to `doc/html/index.html`. Doxygen can pick up comments in the source code and use them to generate descriptive and useful information about the code. To do so, the comments need to be formatted in a specific way, which is explained in great detail on the official Doxygen web-site.

6.2 Debugging and Profiling

Debugging is a common activity for any programmer and is typically necessary for any given development project. The first and easiest method of debugging is adding Info statements to narrow down broken parts of the code. There exists increasingly more sophisticated approaches such as using a dedicated debugging program, such as `gdb`. The process of code profiling, however, is typically only applied to functioning programs with the goal of acceleration and decreasing memory usage.

6.2.1 Debugging with GNU Debugger (`gdb`)

Debugging code using GNU Debugger (`gdb`) is more beneficial when OpenFOAM has been compiled without any optimizations. Optimizations are performed by the compiler and generally do not interfere with the debug information generated for the compiled code. However, profiling and examining code that is not optimized by the compiler may provide more insight into hidden computational bottlenecks.

Compiler flags used for OpenFOAM are bundled up into groups of options which can be interchanged depending on the target configuration. The compiler options are found in the `$WM_PROJECT_DIR/etc/bashrc` global configuration script and are listed in listing 31.

Listing 31 Compiler configuration flag options.

```
#- Optimised, debug, profiling:
# WM_COMPILE_OPTION = Opt | Debug | Prof
export WM_COMPILE_OPTION=Opt
```

To enable working with OpenFOAM in debugging mode using `gdb`, the `$WM_COMPILE_OPTION` variable must be set in the following way:

```
export WM_COMPILE_OPTION=Debug
```

In order for this change to take effect, OpenFOAM must be re-compiled. Re-compilation is also necessary for all custom libraries and applications which require debugging. Please bare in mind that the execution time of code compiled with the debug option active is significantly longer.



TIP

Debugging with gdb is quite straightforward and there is plenty of information on working with gdb available on the official website www.gnu.org/software/gdb/.

Debugging the code can sometimes be simple for bugs such as a segmentation fault (accessing the wrong memory address) or a floating point exception (dividing by zero). Those errors in the execution of the code trigger system signals, such as SIGFPE (floating point exception signal) and SIGSEV (segmentation violation signal) and are trapped by the debugger. The debugger then allows the user to browse through the code in order to find the error, set break points, examine variable values and much more.

In order to show gdb in action, an example for debugging with gdb is covered in this section. This tutorial is provided in the sample code repository in the `primer-examples/applications/test/testDebugging` directory. The tutorial consists of a testing application that contains a function template used for computing a harmonic mean of a given field, over all time steps of the simulation. Because the harmonic mean involves computing $\frac{1}{x}$, where x is the field value, a floating point exception (SIGFPE) appears if the code is executed on a field which contains zero values. The function template is defined in the `fvc::` namespace and it behaves similar to the rest of the OpenFOAM operations. It does, however, have a slightly reduced functionality. For this reason, the function template definition and declaration are packed together with a testing application which is not (and should *never* be) a standard practice.

The tutorial simulation case that can be used with this example is the case `primer-example-cases/chapter4/rising-bubble-2D`. Executing the `testDebugging` application with the `-field alpha1` argument within the `risingBubble-2D` case directory results with the following error:

```
?> testDebugging -field alpha1
(snipped header output)
Time = 0
#0 Foam::error::printStack(Foam::Ostream&) at
```

```
/OpenFOAM-2.2.x/src/OSspecific/POSIX/printStack.C:221
#1 Foam::sigFpe::sigHandler(int) at
/OpenFOAM-2.2.x/src/OSspecific/POSIX/signals/sigFpe.C:117
#2 in "/usr/lib/libc.so.6"
#3
at
/primer-examples/applications/ \
test/testDebugging/testDebugging.C:79
(discriminator 1)
#4
at /primer-examples/applications/ \
test/testDebugging/testDebugging.C:217
(discriminator 1)
#5 __libc_start_main in "/usr/lib/libc.so.6"
#6
at ????:?
Floating point exception (core dumped)
```

The first step in debugging this problem is to start `gdb`, combined with `testDebugging`. All components of `testDebugging` must be compiled in debug mode, this also includes all libraries of interest as well as the OpenFOAM platform.

```
?> gdb testDebugging
```

This leads to the console version of `gdb`, which which used to execute any programs which require debugging:

```
(gdb)
```

Within this debugging console of `gdb` any command can be executed for debugging purposes when prepended by the `run` command. For the `testDebugging` application, this means that the `alpha1` field must also be passed as an additional argument:

```
(gdb) run -field alpha1
```

After the execution of the above command, the `SIGFPE` error appears again, but with much more details:

```
Program received signal SIGFPE, Arithmetic exception.
0x0000000000414706 in Foam::fvc::harmonicMean<double>
(inputField=...) at testDebugging.C:79
79          resultField[I] = (1. / resultField[I]);
(gdb)
```



Due to the compilation of OpenFOAM and the application in Debug mode, gdb points directly to the line in the source code which is responsible for the SIGFPE.

TIP

Programming in the Debug mode and using a debugger gives the programmer the ability to find errors in a program much faster compared to using output statements.

For the very basic example of the `testDebugging` application, analyzing the source code manually may lead to the same insight. With increasing algorithm complexity, it becomes less likely that a manual search for a bug will be successful. While inserting Info statements is a popular method to help the manual debugging process, it tends to be a very time consuming process. The debugger, on the other hand, can: use information stored on the memory stack to step into functions, execute loops one iteration after another, change variable values, post break points in execution, condition the break points based on variable values, and many other advanced options you can find in the debugger documentation.

To review a basic workflow with gdb, the aforementioned error is assumed to occur inside a convoluted code-base. This code is located in a library with non-cohesive classes which are strongly coupled with fat interfaces and member functions that span hundreds of lines. In this situation, the programmer needs to examine the code above and below the signal line to get a hold on the context of execution. The debugger can display the path that the signal has taken: from the top-level call in the testing application, all the way down to the low level container that raises this error. This information can be obtained within gdb, by executing `frame` in the gdb console:

```
Program received signal SIGFPE, Arithmetic exception.
0x0000000000414706 in Foam::fvc::harmonicMean<double>
(inputField=...) at testDebugging.C:79
79          resultField[I] = (1. / resultField[I]);
(gdb) frame
#0 0x0000000000414706 in Foam::fvc::harmonicMean<double>
(inputField=...) at testDebugging.C:79
79          resultField[I] = (1. / resultField[I]);
```

Since the `testDebugging` example solely consists of the `main` function, a single stack frame is available, where the information regarding the function object code is stored. When multiple function calls are made, there are multiple frames available. The lowest frame of the `SIGSEV` signal usually leads somewhere to the basic OpenFOAM container `UList`. It is very unlikely, however, that the error is caused by `UList`, and much more likely that it is caused by the custom code. In this case, an addressing error in a container (which inherits from `UList`) is the reason for the error.

Choosing frame 0 and listing the source code with the `list` command, narrows down the location of the error:

```
(gdb) frame 0
(gdb) list
75 volumeField& resultField = resultTmp();
76
77 forAll (resultField, I)
78 {
79     // SIGFPE.
80     resultField[I] = (1. / resultField[I]);
```

Hence the culprit for the `SIGFPE` signal is line 80. To put a break point at line 80, the `break` command must be executed and `testDebugging` must be re-run:

```
(gdb) break 80
Breakpoint 1 at 0x4146cd: file testDebugging.C, line 80.
(gdb) run
The program being debugged has been started already.
Breakpoint 1, Foam::fvc::harmonicMean<double> (inputField=...) at
testDebugging.C:80
80         resultField[I] = (1. / resultField[I]);
```

After having placed the break point in line 80, the variable `I` can be evaluated at this position:

```
(gdb) print I
$1 = 0
```

Printing `resultField[I]` shows that it's value is 0. In order to check the influence of different values of `resultField[I]`, they can be set manually using `gdb`:



```

(gdb) set resultField[I]=1
(gdb) c
Continuing.

Breakpoint 1,
Foam::fvc::harmonicMean<double> (inputField=...)
at testDebugging.C:80
80resultField[I] = (1. / resultField[I]);
(gdb) c
Continuing.

Program received signal SIGFPE, Arithmetic exception.
0x0000000000414706 in
Foam::fvc::harmonicMean<double> (inputField=...)
at testDebugging.C:80
80 resultField[I] = (1. / resultField[I]);
(gdb) print I
$2 = 1
(gdb)

```

For this simple example, it is obvious that at least two values of `I` lead to 0 for `resultField`. As, by definition, the field `resultField` must never be 0, the next step is to investigate if the field has been preprocessed properly.

WARNING

The code in the example code repository might be changed with time, so the actual line numbers in the debugger output may vary slightly from those shown in this text.

TIP

The solution to problem is already included in the sources for this example - uncommenting the marked lines allow the application to run correctly.

6.2.2 Profiling with valgrind

Profiling¹ code with the `valgrind` profiling application is relatively straightforward. `valgrind` allows a developer to find computational bot-

¹According to wikipedia (en.wikipedia.org/wiki/Code_profiling), “profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls”

tlenecks in code as well as displays them graphically using call-graphs and similar diagrams. This information can then be used to concentrate efforts more efficiently when optimizing code for efficiency. Typically, performance is distributed using an estimated 90-10 or 80-20 rule, which means that 90 (80) % of the computation is usually performed by 10 (20) % of the code.

Similar to the previous section on debugging with gdb, the code needs to be compiled with the Debug compilation option. The high-level form of optimization is directly related to software design: choosing algorithms and data structures appropriately. Once the algorithms and data structures are chosen efficiently, there is a possibility of performing what is called low-level optimizations (e.g. loop unrolling).

Complexity is a parameter which used to compute the efficiency of algorithms and data structures. It is denoted by the capital "O": e.g. $O(n \log n)$, where n stands for the number of elements operated on. It generally is not advisable to delve into low-level optimization when there is an alternative algorithm available with a lower complexity. The same applies to data structure selection: choosing data structures carefully is an absolute must. More information of the container structures can be obtained from documentation on C++ data structures present in the Standard Template Library (STL) (see Josuttis (1999) for details). OpenFOAM containers are not STL-based (although some of them provide STL compliant iterator interfaces), but are very much alike in how they function, for example:

DynamicList is similar to `std::vector`. Both have direct access of elements with $O(1)$ complexity and $O(1)$ insertion complexity at the end of the container, if the container size is smaller than its capacity. If the resulting size from an insertion operation is greater than the current capacity, the insertion complexity will be proportional to $O(n)$.

DLList is similar to `std::list`. The insertion complexity (anywhere in the container) is $O(1)$ and the access complexity is $O(n)$.

A lot of the algorithmic work present in OpenFOAM is related to the specific owner-neighbour addressing of the unstructured finite volume mesh, with the advantage of automatic parallelization of the algorithm



code. For standard OpenFOAM library code, the choice of data structures falls from four main container families:

1. List
2. DynamicList
3. A linked list *LList
4. An associative map implemented as a HashTable

Before selecting a container for a specific algorithm, it is important to examine the container interface beforehand. Otherwise the selected container may not suit the needs of the algorithm properly, leading to longer execution times.

The example application `testProfiling` can be used to make this point for the `DynamicField` container using a very simple example. Try executing the profiling test application anywhere on your system (it doesn't require an execution within an OpenFOAM simulation case directory):

```
?> testProfiling -containerSize 1000000
```

The `-containerSize` option passed to the `testProfiling` testing application sets the number of elements that are appended to the two different `DynamicField` objects: the first object is null-initialized, and the second one is initialized with the size that corresponds to the integer value passed to the application. Timing code has been added to the application for the part of the code that does the appending of elements at the end of the `DynamicField`. You can execute the application providing different values for the size of the container and see how the times differ between the initialized container and the container with an initialized size. The `DynamicField` has, as the `DynamicList` container, constant complexity in inserting elements at the end of the container *if the size of the container is smaller after insertion than the initial container capacity*, otherwise the complexity of an insertion at the end (appending) operation is linear with respect to the container size. This means it will have a larger impact on the efficiency of your code if the container sizes are large and even more if the container stores complex objects which then add additional burden of unnecessary $n - 1$ constructions for every element that is appended at the end of the container.

Once you have built the example source code repository in *Debug mode*, you can profile the `testProfiling` application using `valgrind`:

```
?> valgrind --tool=callgrind testProfiling -containerSize 1000000
```

Valgrind utilizes various tools such as catching cache misses, checking the programs for leaked memory, etc, but this is out of the scope of this book. More information on those topics can be found in the official valgrind documentation.

Executing the above command produces an output file called `callgrind.out.ID`, where `ID` is the process identification number. The output file may be opened with `kcachegrind`, which is an open source application used to visualize the output of `valgrind`. As you might have noticed yourself in the timing output produced by the `testProfiling` application, `valgrind` shows that around 62 % of the total execution time is spent for appending the elements to the un-initialized `DynamicField` object, and around 14% is spent on appending elements to pre-initialized `DynamicField` object.

This brings us to conclusion that although `DynamicList` and `DynamicField` are dynamic, there is a cost to pay for appending elements at the end of such a container once its capacity is exceeded. If our algorithm doesn't require direct access to elements, if it loops over the elements one after another, then using a heap-based linked-list might prove to be a better choice. Allocating non-sequential memory blocks and pointer indirection also cost CPU cycles, however, and as such it becomes difficult to know the proper choice of a container in advance. The answer to this problem comes in the form of generic programming, separating algorithms from containers in a careful and thoughtful way (it is not always possible), and profiling the code.

6.3 Using git to Track an OpenFOAM Project

Git is a distributed VCS which is very popular in the open source community. The development was started by Linus Torvalds to track the source code of the linux kernel as he found all existing tools (not mercurial, though) as inappropriate and not capable of doing that job properly.



The benefit of using git is that it is a distributed system. This means that no central server is required. You can create a repository in any directory of your filesystem. If you clone an existing repository (from github, for example) you do not get the latest snapshot only, but the entire repository including its history. All operations are generally executed locally and do not require the remote repository to be available, which is quite handy when working without an internet connection.

It would be beyond the scope of this book to go into details of git and we hence refer to the git documentation ² for more details about git. A reasonable understanding of git is assumed for the remaining chapter. A great branching model can be found on nvie.com³. In the following we try to outline a couple different cases where git can be a reasonable help for working with OpenFOAM.

Getting the Latest Release of OpenFOAM

The stable releases of OpenFOAM (2.2, e.g.) can be obtained as tarballs from the OpenFOAM website⁴ and they are not under version control. Bugfixes, enhancements and new features for the current OpenFOAM version are provided via a git repository. This "rolling" version is always named by its version number (2.2) followed by a ".x", that indicates an arbitrary subversion. This version can be downloaded from github.com/OpenFOAM/OpenFOAM-2.2.x into the current working directory with the following command:

```
?> git clone git@github.com:openfoam/openfoam-2.2.x.git
```

After the command is finished, the entire history is available on your local machine and the history of the release can be investigated. The history, or log, gives a short overview over who changed what and when.

```
?> git log
```

This freshly cloned repository can be compiled just in the same way as the stable release would be.

²<http://git-scm.com/book>

³<http://nvie.com/posts/a-successful-git-branching-model>

⁴openfoam.com

Putting Custom Sources under Version Control

The most common way of using git for custom developments is to track each of your projects in a *seperate* repository that is unrelated to the main release. This makes sharing your code easy, not the entire OpenFOAM release has to be provided alongside with the code. Say three projects should be put under version control: `projectA`, `projectB` and `projectC`. Each of these projects' directories has to be entered individually and a git repository has to be initialized in each of them:

```
?> cd projectA
?> git init
```

Each of those repositories is local and empty. Files have to be added to the repository manually. To prevent the `.deb` files and `lnInclude` directories, that are generated at compile time, from polluting the repository, a `.gitignore` file should be added to prevent them from appearing in the repository. This method enables you to share and port your code between different OpenFOAM versions more easily.

Developing in the Main Repository

Though developing directly in the OpenFOAM repository might look a little bit odd at first, it has a couple of advantages over putting the developments into seperate repositories. If administrating a HPC cluster and install a global OpenFOAM version for all users that should all use some custom developments, this way of developing might come in handy. Merging off the master branch and apply changes *only* to this branch is a fairly safe way to integrate custom developments into the OpenFOAM repository. This branch is then gets deployed to the HPC cluster. The `master` branch should be similar to `master` of the main repository and any upstream changes to `master` should get merged into the local development branch, in order to stay up to date. The deployment itself can get automated further using *git hooks*. Migrating developments from one major release to the next one might turn out to be a little bit more tricky than with seperate and independent repositories, though.



Tracking a Case

As git can track any text files, it is not only limited to source code but can also deal with OpenFOAM cases. To do this properly, `writeFormat` in the `controlDict` should be set to `ascii` and only selected files of the case should be tracked. Some files and directories, such as the timestep and processor directories, should not be tracked by git and must hence be excluded by an appropriate `.gitignore` file. This selection is already done by the `gitignore` project on `github.com`⁵ which ignores the mesh and a lot of other files and directories that are not directly related to the case setup.

A good application for tracking a case is when the influence of various parameters on the simulation is to be investigated. This renders copying the same case many times obsolete. Various versions of the case can be highlighted using git tags.

6.4 Installing OpenFOAM on an HPC cluster

In this section some topics are considered when attempting to install OpenFOAM on a HPC cluster. When working on a remote cluster, a whole new system comprised of different hardware, software libraries, and compiler versions has to be expected. Due to these differences, every system will be different and may hold special challenges for installation. With that said, this section will try to outline some of the more generic issues that you may encounter on any particular system instead of covering system particulars. Moving forward it is assumed that the reader has some experience with executing a compiler and setting its configuration as well as using Linux system environment variables. This includes linking include folders and library binaries to a compilation.

6.4.1 Distributed Memory Computing Systems

Distributed memory computing systems are currently the most common HPC systems used for large scale scientific computing and engineering. These systems are comprised of an agglomeration of interconnected compute nodes. Each node will typically consist of a mainboard with multiple

⁵ github.com/github/gitignore

CPUs which share RAM memory access within this node. Currently a typical compute node comprised of between 4 and 16 independent processor cores. Each node is interconnected via a complex network of fast interconnects. These interconnects can range in data transmission speed and type with the current industry standard being the Infiniband® fiber optic interconnect systems which can pass upwards of 50 gigabytes per second between compute nodes.

Inter-node and within-node-inter-processor data communications are typically handled with an implementation of the MPI programming standard. These implementations may be open source such as OpenMPI or closed source commercial versions. Regardless, it is advisable to use the system configured MPI implementation that is supported by the cluster itself. If compiling against the OpenMPI library included in the stock ThirdParty folder that is packaged with OpenFOAM, the solver may still function but will be without considerable parallel optimizations and interconnect support. This could severely hinder parallel speed and scaling and is not recommended.

Without a fully function MPI compilation, the OpenFOAM Pstream library will not compile correctly. The Pstream library acts as an interface between the CFD library and the raw MPI functions, which is necessary for any parallel computations.

6.4.2 Compiler Configuration

Each cluster will usually have an officially supported compiler for use on the system. This could be an open source compiler such as Gcc, but it may also be a pre-configured commercial C++ compiler such as Icc, offered by Intel®. To select which compiler OpenFOAM's compilation system `wmake` will use, the main configuration file (`$WM_PROJECT_DIR/etc/b`) must be changed. The section of the file that concerns compiler selection is shown in listing 32: The first option, `Compiler location`, will dictate whether the system compiler is used or the compiler that came packaged with OpenFOAM in the `ThirdParty` directory is used by `wmake` in any compilation activities. In the case of an HPC installation, the `system` compiler should always be used. The other option will set which compiler to use. There are a lot of options available, but they all may not be installed on the system in question. For example, many Linux based



Listing 32 Changes to \$WM_PROJECT_DIR/etc/bashrc, that alter the compiler

```

#- Compiler location:
#   foamCompiler= system | ThirdParty (OpenFOAM)
foamCompiler=system

#- Compiler:
#   WM_COMPILER = Gcc | Gcc43 | Gcc44 | Gcc45 | Gcc46 | Clang | Icc
export WM_COMPILER=Gcc

```

compute clusters will have Gcc installed somewhere on the system, but the recommended compiler may be a highly optimized and tuned version of Icc. In this case, the compiler should be defined as:

```

#- Compiler:
#   WM_COMPILER = Gcc | Gcc43 | Gcc44 | Gcc45 | Gcc46 | Clang | Icc
export WM_COMPILER=Icc

```

6.4.3 MPI Configuration

As with the compiler, `wmake` needs to be configured to build against the supported MPI implementation specific to the HPC system. This is done in the same `bashrc` file as compiler selection (see listing 33).

Depending on which implementation is selected, OpenFOAM sets a

Listing 33 Changes to \$WM_PROJECT_DIR/etc/bashrc, that alter the MPI configuration

```

#- MPI implementation:
#   WM_MPLIB = SYSTEMOPENMPI | OPENMPI | MPICH | MPICH-GM | HPMPI
#             | GAMMA | MPI | QSMPI | SGIMPI
export WM_MPLIB=OPENMPI

```

different target location for `MPI_ARCH_PATH` and `MPI_HOME`, which is intended to be the location on the cluster file system where these mpi executables, libraries, and header files exist. One of the most common issues with HPC installations is that this directory is not correctly set and the `Pstream` library doesn't compile correctly due to missing MPI libraires and headers.

In this case, the `WM_MPLIB=OPENMPI` setting will instruct `wmake` to use

the OpenMPI version that was packaged with the ThirdParty directory. If SYSTEMOPENMPI is set instead, the script will look to the system to load the OpenMPI library. The script which defines these environment variables depending on the selection, is located in \$WM_PROJECT_DIR/etc/containers/settings.sh, under the Communications library subsection.

There the MPI_ARCH_PATH and MPI_HOME environment variables are set. Unfortunately, the way the variables are set may not be suitable for the specific configuration of the cluster. For example, choosing HPMPI as the MPI library, the script will attempt to load a specific directory (around line 463):

```
HPMPI)
  export FOAM_MPI=hpmpi
  export MPI_HOME=/opt/hpmpi
  export MPI_ARCH_PATH=$MPI_HOME

  _foamAddPath $MPI_ARCH_PATH/bin
```

If the location of the hpmpi folder is not exactly /opt/hpmpi, MPI_HOME will not be set correctly and the error will cascade throughout the script and eventually disrupt the compilation. Thankfully, setting these variables manually is quite easy, assuming the location of the target MPI implementation in the system directory tree is known.

WARNING

The system administrator of the cluster is the first person to contact when problems with OpenFOAM compilation occur on the cluster.

Lets say this cluster supports a specific, *fictional*, MPI implementation called Super-Scaling MPI, *ssmpi* for short. This fictional library, version 1.3, can be found within the directory tree /opt/apps/ssmpi/1.3. Because the script will not know how to configure *wmake* to link to this, the mpi environment variables have to be defined manually in \$HOME/.bashrc (see listing 34). The variables are set accordingly *after* the OpenFOAM bashrc is sourced to ensure any incorrectly loaded values are overwritten. The importance of having this environment variable set is evident when looking at the *wmake* rules for setting MPI compiler flags (see listing 35). In the configuration file for compiling with OpenMPI, these variables are being used to set the paths to header and binary



Listing 34 Definition of a custom MPI version in `$HOME/.bashrc`

```
source ~/OpenFOAM/OpenFOAM-2.2.0/etc/bashrc
export MPI_ARCH_PATH=/opt/apps/ssmpi/1.3
export MPI_HOME=/opt/apps/ssmpi/1.3
```

locations. If an installation is set up with the fictional SSMPI code, a new configuration file would be created, following this naming convention and populated with the appropriate file paths. In the end, all of this setup

Listing 35 Rules for `wmake`

```
?> cat $WM_PROJECT_DIR/wmake/General/mpilibOPENMPI

PFLAGS    = -DOMPI_SKIP_MPICXX
PINC       = -I$(MPI_ARCH_PATH)/include
PLIBS      = -L$(MPI_ARCH_PATH)/lib$(WM_COMPILER_LIB_ARCH)
            -L$(MPI_ARCH_PATH)/lib -lmpi
```

is to ensure that the `Pstream` library correctly compiles and links to the local MPI implementation. The `Pstream` compilation options, stored in `$WM_PROJECT_DIR/src/Pstream/mapi/Make/options`, contain `PFLAGS`, `PINC`, and `PLIBS` being fed directly to the compiler flags via `-I` and `-L`. This will be used at compilation time for header and library linking. Beyond interfacing with an MPI library, OpenFOAM is mostly self contained and has only a few other external library dependencies. If significant trouble occurs during compilation, there is a high probability it occurred during configuration of these MPI libraries and compiler settings.

Unfortunately, any number of issues and errors could arise during compilation on any given system. Often times reaching a successful compile will require a user to draw upon personal experience gained from using compilers as well as working within different operating system environments. Luckily, with the growing usage base of OpenFOAM, many new systems are coming online with a local version pre-installed and configured. If a user has relatively little experience using compilers and requires an HPC cluster to be used with OpenFOAM, it may be a good idea to specifically look to systems with the configured package.

Further reading

Josuttis, Nicolai M. (1999). *The C++ standard library: a tutorial and reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.



7

Turbulence Modeling

This chapter covers the modeling of turbulence in OpenFOAM. The details on the physical and mathematical modeling are kept to a minimum. The reason for this decision lies in the fact that turbulence modeling is a large topic in itself, one beyond the scope of this book. For more in-depth discussions of turbulence modelling, the reader is referred to respective literature, such as Lesieur (2008), Pope (2000), Pozrikidis (2011), and Wilcox (1998).

7.1 Introduction

In general, there are four different approaches to modeling turbulence and its effects. The main purpose of the turbulence models is to determine the Reynolds stresses, as they are unknown in the momentum equation (see chapter 1).

Depending on the model type, the way of calculating the Reynolds stresses can vary from very easy to very complex, which in turn leads to different requirements to the required computational effort and the computational grid. Figure 7.1 provides a compact overview over these models.

The most basic category of turbulence models are the RANS models, as all of models of this group work on the *temporal* fluctuation of the

TIP

The Reynolds stresses can be regarded as “*averaged momentum flow per unit area, and so comparable to a shear stress*” (see Baehr and Stephan 2011) and they introduce additional unknowns to the flow equations. This leads to a *closure* problem, due to the nonlinearity of the Navier-Stokes equations, when they are averaged, which leads to the *Reynolds* equations (more details in Lesieur 2008). To be able to solve this problem, more information must be provided. In this specific case, the Reynolds stresses must be modeled somehow, which is the point where the turbulence models come into play (see Pope (2000)).

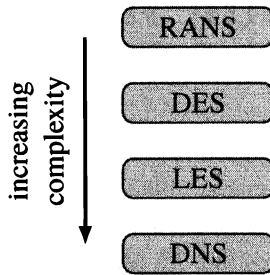


Figure 7.1: Relative computational complexity of various turbulence models and direct numerical simulations.

mean velocity, the *Reynolds averaging* (see Lesieur 2008; Reynolds 1895). Models of this type can work on a comparatively coarse grid, as the turbulent fluctuations are not resolved geometrically, but modelled. If taken literally, these models are only suitable for steady state simulations, as no real turbulent fluctuations can be modeled. Prominent examples for such RANS models are the $k-\epsilon$ model (see Jones and B.E. Launder 1972; B.E. Launder and Sharma 1974), the $k-\omega$ model (see Wilcox 1988) and $k-\omega$ -SST model (see Menter 1993, 1994). The implementations of these models in OpenFOAM are located in the Reynolds Averaged Simulation (RAS) model library.

The next group of models is called LES and differ from RANS models therein that only small scale eddies are modeled. Large scale eddies are resolved spatially by the computational grid, which needs to be signifi-



cantly finer, than for RANS models. In terms of computational costs and efficiency, LES lies directly between RANS and Direct Numerical Simulation (DNS), as shown in figure 7.1. Pope (2000) states that “*LES can be expected to be more accurate and reliable than Reynolds-stress models, for flows in which large-scale unsteadiness is significant*”.

DNS is based on solving the Navier-Stokes equations for all flow scales, without modeling any of them otherwise. Implementation-wise this approach is the simplest one, but the required computational efforts are extremely high.

Turbulence modeling in OpenFOAM is generic, hence any model can be selected within each solver. This of course assumes that the solver supports turbulence modeling, which is the case for most of them. The generic implementation has the major benefit of being able to use RTS in conjunction with the turbulence models and not having to recompile the solver, when a different turbulence model should be used. All turbulence models can be found in `$FOAM_SRC/turbulenceModels` and their respective implementation varies between compressible, incompressible and LES type models. In the following, only a small subset of the RANS models is covered, whereas LES, Detached Eddy Simulation (DES) and DNS models are neglected.

7.1.1 Wall Functions

“*At high Reynolds number, the viscous sublayer of a boundary layer is so thin that it is difficult to use enough grid points to resolve it*” (see Ferziger and Perić 2002). The wall functions rely on the *universal law of the wall*, which states that the velocity distribution very near to a wall is similar for almost all turbulent flows.

One of the most prominent parameters, when it comes to judging the applicability of wall functions is the dimensionless wall distance y^+ , defined by Schlichting and Gersten (2001) as:

$$y^+ = \frac{yu_\tau}{\nu} \quad (7.1)$$

With y representing the absolute distance from the wall and u_τ and ν denote the friction velocity and kinematic viscosity, respectively.

Pope (2000) provides great insight into wall functions and why they are so important. Due to the steep velocity profile, relatively near to the wall, the turbulence model needs to account for that. This is where wall functions come into play, which were firstly suggested by B.E. Launder and Spalding (1972). The idea is that additional boundary conditions are applied at some distance to the wall, to fulfill the log-law. Hence the additional equations introduced by the turbulence model are not solved close to the wall. Depending on the particular turbulence model used, different wall functions must be applied to the respective fields of the turbulence model. Meaning that a $k - \epsilon$ model requires different wall functions than a $k - \omega$ model.

WARNING

Especially when the flow suffers severe flow separation, RANS models are unable to capture this separation properly. Hence such flow problems must be handled with care, if RANS models are used (see Ferziger and Perić (2002), Pope (2000), and Wilcox (1988)).

Different turbulence models and their associated wall functions require different values of y^+ and hence different spatial resolution of the near wall area. The reader is referred to the particular literature, for required y^+ values, that the computational grid near the wall must achieve. Note that if the log-law region is resolved geometrically by the mesh, no wall functions must be applied. Depending on the type of simulation, such low y^+ values are either extremely hard to achieve during the meshing process or even undesirable, as this significantly decreases the time step.

In OpenFOAM, wall functions are nothing else than ordinary boundary conditions, that are applied to boundary patches of type `wall`, rather than the usual `patch`. The only restriction that come along with the wall functions is that the boundaries must suffice the latter requirement. Otherwise the solver will complain with an error message on execution. As wall functions are quite similar to boundary conditions, implementation wise, their design is not discussed explicitly in this chapter. Boundary conditions are discussed at great length in chapter 10.



7.2 Pre- and Post-processing and Boundary Conditions

Depending on the choice of turbulence model, new fields are introduced to the simulation, which need to be solved as well. For the sake of simplicity, the $k-\omega$ turbulence model is selected as an example. There are basically two types of boundary conditions that can be used to model the parameters of the turbulence model at e.g. the inflow of the computational domain.

Standard boundary conditions such as `fixedValue` or `inletOutlet` boundary conditions can be used, if the particular inflow values are known to the user. These values, namely turbulent kinetic energy k and the specific dissipation rate ω can be calculated manually from either the turbulence intensity I or the mixing length L (see Fluent 2005):

$$k = \frac{3}{2} (|\mathbf{U}_{\text{in}}| I)^2 \quad (7.2)$$

$$\omega = \rho \frac{k}{\mu} \left(\frac{\mu_t}{\mu} \right)^{-1} \quad (7.3)$$

$$\omega = \frac{\sqrt{k}}{C_\mu^{\frac{1}{4}} L} \quad (7.4)$$

Here the turbulence intensity should be selected as $I \in [0.01; 0.1]$ and for a freestream flow, $I = 0.05$ is a common choice. The turbulent viscosity ratio $\frac{\mu_t}{\mu}$ at inflow boundaries is assumed to be quite low and usually selected as $1 \leq \frac{\mu_t}{\mu} \leq 10$, as discussed by Fluent (2005). Equation (7.4) is commonly used, when the mixing length is known, whereas equation (7.3) can be used in the other cases, in a straight forward fashion. When these values are determined, they can simply be assigned to the respective boundaries. Still they have to be computed manually.

Turbulence specific boundary conditions can be used for some cases, which in turn implement some of the above equations, which can then be used for the initialisation of the inflow. The first of these special inflow boundary conditions is `turbulentIntensityKineticEnergyInlet`, which initialises k according to equa-

tion (7.2). Applying this boundary condition to a boundary INLET, leads to the code listed in listing 36.

Listing 36 Example of turbulentIntensityKineticEnergyInlet

```
INLET
{
    type    turbulentIntensityKineticEnergyInlet;
    intensity 0.05;
}
```

Additionally, the inlet for ω based on the turbulent mixing length initialization (equation (7.4)), can be directly defined using turbulentMixingLengthFrequencyInletFvPatchScalarField. Similar to turbulentIntensityKineticEnergyInlet, this boundary condition just requires some additional parameters and then computes ω based on equation (7.4). The required dictionary for the boundary named INLET is shown in listing 37. Here, C_μ is read di-

Listing 37 Example of turbulentMixingLengthFrequencyInlet

```
INLET
{
    type            turbulentMixingLengthFrequencyInlet;
    mixingLength    0.005;
}
```

rectly from the selected turbulence model and must not be specified again.

For other fields introduced by other turbulence models, there are equivalent boundary conditions contained in the OpenFOAM framework.

7.2.1 Pre-processing

Some other neat preprocessing applications include boxTurb and applyBoundaryLayer. boxTurb can be used to initialise a non-uniform and turbulent-appearing velocity field, which is convenient for cases where turbulent effects play a major role and must be present from the start of the simulation. The resulting velocity field still suffices the continuity equation and is hence divergence free.

The development of a near-wall flow can be simplified and its convergence improved by using applyBoundaryLayer. It calculates the boundary



layer based on the $\frac{1}{7}$ -th power-law (see Chant 2005; Fluent 2005) and the velocity field is adjusted accordingly.

This tool provides two custom commandline parameters, namely `ybl` and `Cbl`. The desired boundary layer thickness can be specified directly via `ybl`, whereas `Cbl` calculates the boundary layer thickness as the product of its argument and the mean distance to the wall. Optionally the turbulent viscosity field ν_t can be stored to disk, which is not required to be present by the turbulence models.

7.2.2 Post-processing

In conjunction to the previously described preprocessing tools, there are various postprocessing applications in OpenFOAM. After each simulation that employs turbulence modelling, the y^+ values need to be checked. For this purpose, there are `yPlusRAS` and `yPlusLES`. Each of which can be used for RANS or LES simulations, respectively. Their working principle is quite similar and is not to be discussed. To calculate the particular y^+ values, the respective command must be invoked in the simulation case of question. By default, y^+ is calculated for each available time directory of the case. This can be limited to specific ones by passing the `-times` parameter with appropriate arguments or simply the `-latestTime` option. The (shortened) output looks like the following:

```
?> yPlusRAS -latestTime
Calculating wall distance

Writing wall distance to field y

Patch 2 named wall y+ : min: 1304.38 max: 13427.7 average: 8344.93

Writing yPlus to field yPlus
```

As can be seen from the above listing, not only the minimum, maximum and average y^+ values for the patches of type `wall` are print to the screen. Also the wall distance field `y` and y^+ are written to the case directory, so that they can be judged later on, which is quite convenient. In case a LES model is used, `yPlusLES` has to be employed. It works similar to `yPlusRAS` and is hence not addressed explicitly.

Sometimes the Reynolds stresses R must be calculated and written to a field, for postprocessing purposes. The command line tool `R` is tailored to do exactly this, with no additional arguments required.

7.3 Class Design

The design of turbulence models is quite similar to the ones of transport models and is hence not repeated in this section. Transport models are covered in chapter 11. Rather than being able only to access an individual turbulence model directly, the models are nested into sub-categories. As a result, both the sub-category of turbulence models as well as each individual turbulence model can be selected using Runtime Selection (RTS). This leads to the `RASModel` and `LESModel` type of sub-categories, each of them containing turbulence models.

Further reading

- Baehr, H.D. and K. Stephan (2011). *Heat and Mass Transfer*. Springer.
- Chant, Lawrence J. De (2005). “The venerable 1/7th power law turbulent velocity profile: a classical nonlinear boundary value problem solution and its relationship to stochastic processes”. In: *Applied Mathematics and Computation* 161.2, pp. 463–474.
- Ferziger, J. H. and M. Perić (2002). *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer.
- Fluent (2005). *Fluent 6.2 User Guide*. Fluent Inc. Centerra Resource Park, 10 Cavendish Court, Lebanon, NH 03766, USA.
- Jones, W.P and B.E Launder (1972). “The prediction of laminarization with a two-equation model of turbulence”. In: *International Journal of Heat and Mass Transfer* 15.2, pp. 301–314.
- Launder, B.E. and B.I. Sharma (1974). “Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc”. In: *Letters in Heat and Mass Transfer* 1.2, pp. 131–137.
- Launder, B.E. and D.B. Spalding (1972). *Mathematical Models of Turbulence*. Academic Press.
- Lesieur, M. (2008). *Turbulence in Fluids*. Fluid Mechanics and Its Applications. Springer.
- Menter, F. R. (1993). “Zonal two-equation $k - \omega$ turbulence models for aerodynamic flows”. In: *AIAA Journal*, p. 2906.



- (1994). "Two-equation eddy-viscosity turbulence models for engineering applications". In: *AIAA Journal* 32.8, pp. 1598–1605.
- Pope, S. (2000). *Turbulent Flows*. Cambridge University Press.
- Pozrikidis, C. (2011). *Introduction to Theoretical and Computational Fluid Dynamics*. 2nd ed. Oxford University Press.
- Reynolds, O. (1895). "On the Dynamical Theory of Incompressible Viscous Fluids and the Determination of the Criterion". In: *Philosophical Transactions of the Royal Society of London. A* 186, pp. 123–164.
- Schlichting, Hermann and Klaus Gersten (2001). *Boundary-Layer Theory*. 8rd rev. ed. Berlin: Springer.
- Wilcox, D. C. (1988). "Re-assessment of the scale-determining equation for advanced turbulence models". In: *American Institute of Aeronautics and Astronautics Journal* 26.
- (1998). *Turbulence Modeling for CFD*. D C W Industries.



8

Writing Pre- and Post-processing Applications

Although there are many Pre- and Post-processing applications distributed with OpenFOAM, running specific simulations as well as programming and using new algorithms may require the user to develop new applications. This often leads to tailor-made applications that are only used for a specific purpose.

Before the development of a new Pre- or Post-processing application is considered, the user should make sure that an alternative similar application is not distributed along with OpenFOAM, which might accomplish the task in question without much programming effort.

For example, a user friendly and advanced application for preprocessing was developed by Berndhard Gschaider as a part of his project `swak4foam`, namely the `funkySetFields` application. This application implements an algebraic `exfunkySetFields` parser for OpenFOAM fields. The expression parser extracts arithmetic and differential operations executed on fields from user defined expressions, and evaluates them using the numerical libraries of OpenFOAM. It is highly unlikely that a new application needs to be developed from scratch for general purpose

field arithmetics. In the case where a necessary feature is missing, it is likely simpler to extend the `funkySetFields` with new calculation than to program and test the new application from scratch.

If an appropriate application cannot be found among community contributions or native releases, the application must unfortunately be programmed from scratch. In this case, it is typically good practice to find an existing application with similar functionality and modify it to fit the required task. As covered in chapter 6, the Doxygen generated HTML documentation can help identify and locate those parts of the OpenFOAM framework that can be used to build the new application.

8.1 Code Generation Scripts

OpenFOAM provides a set of Linux shell scripts that help with creating skeleton source code files: application, class, class templates, and build files (used by the `wmake` build system). When creating a simple pre- or post-processing application, only a single source code file is typically used (e.g. `applicationName.C`). Following the OpenFOAM naming convention, it should be stored in the directory which is named in the same way as the application.

To begin programming a new application, create a new directory named the same way as the application. Afterwards the `foamNew` script is executed within the directory, as presented in listing 38.

Listing 38 Initialization of a new application

```
?> mkdir applicationName
?> cd applicationName
?> foamNew source App applicationName
foamNewSource: Creating new interface file applicationName.C
wmakeFilesAndOptions: Creating Make/files
wmakeFilesAndOptions: Creating Make/options
```

It is a good choice to rename the variable `$FOAM_APPBIN` to `$FOAM_USER_APPBIN` in the file `Make/files`. Otherwise the installation process will copy the compiled binary files into the directories usually reserved for binary files of the applications distributed with OpenFOAM. Populat-



WARNING

The code generation scripts generate the official OpenFOAM source code header, that assigns the copyright to the OpenFOAM Foundation. The copyright line can be modified by the programmer if he/she so decides.

ing the OpenFOAM binary folders is a bad practice because it pollutes the directories reserved for the applications distributed with OpenFOAM. Sometimes such compilation is necessary. For example, if a system is used that has OpenFOAM installed in a directory that requires root user access rights, and applications are to be made available to all system users. To start working on the new application, the file `applicationName.C` is to be edited and afterwards compiled with the OpenFOAM build script `wmake` called within the directory `applicationName`:

```
?> wmake
```

The `Make/options` file holds the list of directories where the declarations are stored, and possibly definitions in cases when class/function templates are used. The directories that are to be searched by the `wmake` build system are set with the `-I` option:

```
-I$(LIB_SRC)/finiteVolume/lnInclude
```

In cases when the `-I` option does not use the absolute path, environmental variables are used, such as `$(LIB_SRC)` in the example above. Additionally, a list of directories that hold precompiled dynamic libraries that are linked with the application is defined. The directories that contain the libraries are appended to the build options with the option `-L`:

```
-L$(LIBRARY_VARIABLE)/lib
```

In this example, `LIBRARY_VARIABLE` is an environmental variable which stores the path to the package `lib` folder. This is the location where loadable library binary files are stored. When an application is composed from additional libraries, the `Make/options` file needs to list the libraries to be linked against:

`-lusedLibrary`

Here the `usedLibrary` library will be linked against our compiled application at runtime.

The steps described above are application specific so they will be described in more detail in the following sections. More information about libraries, linking, and the build process can be found in any book about programming in the Linux environment and on the Internet.

8.2 Custom Pre-processing Applications

Due to the functional breadth of existing pre-processing applications, there is rarely the need to create entirely custom OpenFOAM applications. A surprising number of tasks can often be executed and automated using a variety of applications chained together via shell or Python script. The convention in OpenFOAM is that this type of script be named `Allrun` and stored in the directory of the simulation case.

This section covers examples which range in complexity. In one example, a simple shell script calls OpenFOAM executables with while utilizing existing pre-processors. In a second example, the PyFoam library helps generate a parameter study. The PyFoam project is primarily developed by Bernhard Gschaider - it consists of a set of libraries and executable programs written in the Python programming language. Easy parametrization and result analysis of OpenFOAM simulations are the main goals of the PyFoam project. More information about PyFoam can be found on the [foam-extend Wiki Page](#).

8.2.1 Decomposing and Starting a Parallel Run

Consider a situation where a high performance computer is to be used to start a large number of simulations. For this example, all simulation cases will be decomposed in different sub-domains and stored in a directory named `simulations`. The `simulations` directory is to be placed as a sub-directory of the `$FOAM_RUN` directory. Instead of manually starting each simulation, it is preferable to automate this process as much as possible. For this purpose, a shell script can to be programmed through



the following steps:

1. Retrieve the solver name from `controlDict`.
2. Retrieve the number of sub-domains from `decomposeParDict`.
3. Execute the `mpirun` command with the correct number of processors.

In this example, the script is named `Allparrun` and is stored in the `simulations` directory. As a first step, a few lines should be added to the beginning of the script. Similar code is found in most of the tutorials that are distributed along with the release of OpenFOAM:

```
#!/bin/bash
cd ${0%/*} || exit 1

source $WM_PROJECT_DIR/bin/tools/RunFunctions
application=`getApplication`
```

The `getApplication` function is an existing bash function in OpenFOAM and fits in nicely for this example as the name of the application will be the name of the solver in this case.

Next, all case sub-directories in the `simulations` directory must be located. It is assumed that those directories are all proper OpenFOAM simulation cases. As a consequence, an OpenFOAM simulation will be started in each sub-directory of the `simulations` directory. The `find` command is used to look specifically for directories that can be found in `$FOAM_RUN/simulations` (see listing 39). The above lines look for

Listing 39 Looping over OpenFOAM cases, using bash

```
for d in `find $FOAM_RUN/simulations -type d -maxdepth 1`
do
    # This part will be programmed at a later point in this section.
done
```

any directory that is located directly (not recursively) in the `$FOAM_RUN` directory so that the same actions can be performed on each of these cases later on. The `find` command can be easily enhanced by e.g. filtering for certain directory names using wildcards, but for the sake of simplicity it is assumed in this example that the `simulations` directory solely contains OpenFOAM cases.

The next step is to retrieve the number of sub-domains from `system/controlDict` and store this in a variable for later use. For this purpose a new function is introduced into the `Allparrun` script called `nProcs` which returns the number of sub-domains:

```
function nProcs
{
    n=`grep "numberOfSubdomains" system/decomposeParDict \
        | sed "s/numberOfSubdomains\s//g" \
        | sed "s/;/g"`
    echo $n
}
```

The function calls the `grep` program to filter for the line in `system/decomposeParDict` that contains `numberOfSubdomains`. In this line the 'numberOfSubdomains' string is deleted, any whitespace between this keyword and the actual numerical value is removed, and the trailing semicolon is deleted. The result of this chained command is stored in the bash variable `n`. To return it, `echo` is used.

Running any OpenFOAM solver or utility in parallel is done using the `mpirun` command. For this example, this means that two variables need to be passed over to `mpirun` as option arguments: the solver name, and the number of processors. The completed `Allparrun` script is shown in the listing 40.

WARNING

Keep in mind that in this case the call to `mpirun` is starting all jobs as *local* jobs on the machine. In order to initiate it on an HPC cluster and use its compute nodes, other steps are necessary that depend on the specific cluster configuration and are therefore omitted here.

For the most simple case, where a job queuing and submission system is not available, you'll only have to provide a machine file to `mpirun`. The machine file contains either the IP addresses or the host names of the computing nodes on the HPC cluster network. Refer to the respective users manuals provided by your HPC cluster administration team for further information on this topic.



Listing 40 Finished Alparrrun script used for simple automated execution of a number of simulations.

```
#!/bin/bash
cd ${0%/*} || exit 1

source $WM_PROJECT_DIR/bin/tools/RunFunctions

function nProcs
{
  n=`grep "numberOfSubdomains" system/decomposeParDict \
    | sed "s/numberOfSubdomains\s//g" \
    | sed "s/;/;/g"`
  echo $n
}

PWD=`pwd`

for d in `find $FOAM_RUN/simulations -type d -maxdepth 1`
do
  # Jump into the current case directory
  cd $d
  n=`nProcs`
  application=`getApplication`

  runApplication decomposePar

  # Depending on the environment of your HPC, this must be
  # adjusted accordingly. Remember to provide a proper
  # machine file, otherwise all jobs will be started on the
  # master node, which is undesirable in any way.
  mpirun -np $n $application -parallel > log &

  # Jump back
  cd $PWD
done
```

8.2.2 Parameter Variation using PyFoam

In this example, a simulation involving a two-dimensional NACA0012 airfoil is prepared. Mesh generation is performed using `blockMesh` and `mirrorMesh`. The parametric study itself investigates the influence of the angle of attack α on the lift and the drag force. Overall, 15 simulations need to be run with a varying angle of attack from $\alpha = 0^\circ$ to $\alpha = 15^\circ$. Later on, the peak in the graph of the lift force is resolved more precisely by creating new simulations automatically after the first 15 are finished. Without automation, extensive parameter variation in CFD projects can quickly become a laborious and error prone activity.

The modular nature of OpenFOAM is based on different single-task applications and text-based input files as the main data format. As a consequence, OpenFOAM is better suited for automation than many GUI-based CFD platforms. In non-GUI interfaces the user can independently choose how to implement the execution and parameter variation commands in a scripting language of their choice. Using a shell script for automation is indeed possible but sometimes requires the user to write as well as learn a large amount of BASH code. As an alternative, the Python scripting language can often be used in place of many BASH codes. Additionally, many Python based libraries already exist with functionalities designed to support numerical simulations such as interpolation, root finding, data processing, and visualization. This may greatly reduce the required effort to automatically parametrize OpenFOAM simulations and analyze their output. Luckily, Bernhard Gschaider went through the trouble of developing an entire Python library that handles various actions and communications between Python and OpenFOAM: PyFoam.

TIP

PyFoam only interacts with OpenFOAM by means of working with configuration and output files - no modification of OpenFOAM is required.

There are many executables within PyFoam that can be executed from the command line. One example is `pyFoamClearCase.py` which clears the OpenFOAM simulation case directory of previously computed results. The executables are built upon the PyFoam library that is exploited in the following example. Using PyFoam libraries in your Python scripts enables you to access and handle any OpenFOAM case such as alter its structure, access dictionaries easily, work with fields, etc. Additionally, it is possible to execute OpenFOAM executables from within Python in a straightforward manner.

For this example we assume that you have a background knowledge of Python and that PyFoam is installed on your machine properly.

As PyFoam is a Python library that does not modify OpenFOAM in any way, the installation of PyFoam is quite easy. There are some restrictions to PyFoam - especially concerning handling large data files that can be



WARNING

There is ample information available on how to install and configure PyFoam on the OpenFOAM-extend Wiki. It is advised to look up and follow through a tutorial on Python language basics before proceeding with the example in this section.

generated as output of large scale CFD simulations.

There is another Python-based library called PythonFlu¹ which aims to reduce the complexity of the C++ OpenFOAM interface and provide interactive work with OpenFOAM using python. The PythonFlu library relies on a wrapping generator in order to bring interactivity in working with the OpenFOAM interface, which would otherwise be impossible because the compilation of the C++ code is required. More information about PythonFlu can be found on the project website.

For the method presented in this section, a directory needs to be prepared that holds two things: the Python script that takes care of the parameter variation, and a template simulation case. In this example the Python script is named `parametricStudy.py` and the template case is named simply `template`.

WARNING

It is essential that the template case is ready to run and set up properly, as all parametrized cases are created from it.

Before going into details about the script, the necessary background information is provided about the simulation. A symmetric mesh is generated using `blockMesh`, that consist of three blocks as outlined in figure 8.1. The grading of the blocks is adjusted so that there is no noticeable change in cell sizes from block 1 to 2 and from 2 to 3. To resolve the viscous boundary layer along the airfoil fairly small cells are generated near the NACA profile.

All patches are named according to labeling shown in figure 8.1. The

¹<http://pythonflu.wikidot.com/>

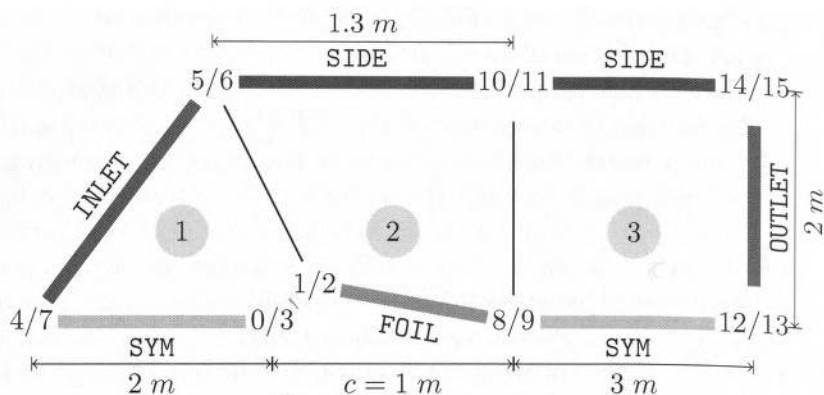


Figure 8.1: Overview of the NACA mesh

vertex numbers are indicated by the two numbers separated by a dash. All edges on the FOIL patch that are not oriented in the normal direction of the figure are shaped using the `polyLine` of `blockMesh` with the station being calculated according to the function that defines any two digit NACA profile. The same goes for the vertices of the blue INLET patch - but rather than using the two digit NACA polynomial, a quarter circle is used. This makes the mesh look more visually appealing and removes any boundary condition issues between the INLET patch and the SIDE patch.

All dimensions of the computational domain as well as the NACA profile itself can be obtained from figure 8.1. The employed solver is `simpleFoam` and will be executed in serial mode. For more details about the boundary conditions and any other data, please have a look at the case provided in the `primer-example-cases` repository.

WARNING

The provided case is for demonstration purposes only. It has been modified to reduce the required computational time.

At this point the programming of the Python script can begin with importing necessary modules, displayed in listing 41. The first three lines import three important components of the PyFoam library:



Listing 41 Imported packages

```
import PyFoam.RunDictionary as RunDict
from RunDict.SolutionDirectory import SolutionDirectory
from RunDict.ParsedParameterFile import ParsedParameterFile
from PyFoam.Basics.DataStructures import Vector
from numpy import linspace,sin,cos, ndarray
from os import path, getcwd
```

- the `SolutionDirectory` that handles an entire case and provides easy access to its data as well as methods to clone the case to a different directory,
- the `ParsedParameterFile` that reads any OpenFOAM dictionary and provides access to it in a pythonic way, using Python dictionaries, lists and tuples,
- the `Vector`, that simplifies writing a vectorial property.

After those PyFoam related imports, some components of the `numpy` module are imported as well as the `os` package, to simplify the code. The imports are followed by some required definitions such as the free-stream velocity and the angles of attack:

```
U = ndarray([1.0,0,0])
alpha = linspace(0,15,15)
```

At the beginning of the script, the `template` case must be read as `SolutionDirectory` and then cloned for each angle of attack. Constructing the `SolutionDirectory` simply requires the path to the case that is going to be read in as show below:

```
template = SolutionDirectory(path.join(getcwd(), "template"))
```

Now we are ready to clone the `template` case for each angle of attack. The `cloneCase` method returns a new `SolutionDirectory` that points to the cloned case, which comes in handy. It is therefore is assigned to a variable for additional optional later processing.

```
for alphaI in alpha:
    name = "alpha%.2f" % alphaI
    currentCase = template.cloneCase(path.join(getcwd(), name))
```

The next step is to read in the velocity boundary condition and write

WARNING

All following code snippets are part of the loop and follow after the line that takes care of the cloning process.

the inflow velocity according to the particular `alphaI`, which is shown in listing 42. The first three lines rotate the velocity vector according

Listing 42 Modify a boundary condition using python

```
uI = U
uI[0] = U * cos(alphaI)
uI[1] = U * sin(alphaI)
uFilePath = path.join(currentCase.name, "0", "U")
uFile = ParsedParameterFile(uFilePath)
movingWallU = uFile[boundaryField]["movingWall"]["value"]
movingWallU.setUniform(Vector(uI[0], 0, uI[1]))
uFile.writeFile()
```

to the current angle of attack. After that, a `ParsedParameterFile` is constructed for the velocity boundary condition `U`, using the path to the current case that can be accessed via `SolutionDirectory.name`. Setting a value in a dictionary is fairly straightforward, as the next line shows. Nested `OpenFOAM` dictionaries are represented as Python dictionaries and can be accessed by their respective names. A uniform value can be set for an entry by using the `setUniform` method. As the simulation is still three-dimensional, we have to set the y velocity component to zero. This is basically everything that is essential to generate a parametric study using `PyFoam`. Executing each case can either be achieved using `PyFoam`'s `BasicRunner` or simply using a shell script.

8.3 Custom Post-processing Applications

Custom post-processing applications are programmed when no similar application can be found that performs the required calculation, which is usually the case when the calculation is very specific. This section covers the programming of a post-processing application that computes the velocity of a rising bubble using an iso-surface computed from the volume fraction field.

In `OpenFOAM`, an iso-surface is represented as a surface mesh consisting



WARNING

There are other ways of performing this calculation, but using the iso-surface shows how to program new classes, re-use existing ones and make use of the new class library in a new post-processing application.

of triangular elements. In this example, the geometry of that triangular surface mesh is used to compute the position of the bubble in each time step. The bubble velocity as well as bubble area are also calculated, which may be useful for certain validation cases.

The rising bubble example simulation case (`rising-bubble-2D`) from the `primer-example-cases` repository is used in this example. The example application is named `isoSurfaceBubbleCalculator` and it is available in the `primer-examples` repository. If the `primer-examples` repository has been configured properly, the application source code can be found in the `isoSurfaceBubbleCalculator` directory, placed in the

```
$PRIMER_EXAMPLES/applications/utilities/postProcessing/
```

folder. The supporting library is named `bubbleCalc` and its source code is placed in the

```
$PRIMER_EXAMPLES_SRC/bubbleCalc
```

directory.

In the following text, the implementation of the example `bubbleCalc` library is described. Additionally, the `isoSurfaceBubbleCalculator` post-processing application is analyzed using source code listings.

WARNING

The source code listings show only segments of the class/application code, the library source code files should be opened in a text editor for a complete overview of the implementation.

Because the bubble will be described with an iso-surface triangular mesh,

the `isoSurface` class will be the starting point of the implementation. The implementation of the `isoSurface` class can be found in the `isoSurface.H` file in the `$FOAM_SRC` directory using the `locate` shell command. Alternatively, the class declaration can be found by searching for the class name using the search box of the HTML documentation generated with Doxygen. Information about generating and using the HTML documentation of the source code with Doxygen is covered in chapter 6. The interface of the `isoSurface` class shows that the complex construction of the iso-surface triangle mesh is placed in the constructor of the class, as shown in listing 43.

Listing 43 Constructor declaration of the `isoSurface` class.

```
// - Construct from cell values and point values. Uses boundaryField  
// for boundary values. Holds reference to cellIsoVals and  
// pointIsoVals.  
isoSurface  
(  
    const volScalarField& cellIsoVals,  
    const scalarField& pointIsoVals,  
    const scalar iso,  
    const bool regularise,  
    const scalar mergeTol = 1E-6    // fraction of bounding box  
);
```

The iso-surface mesh building algorithm is placed in a constructor and not a member function, and no empty constructor is provided by the class. As a consequence, the only way to reconstruct the iso-surface is to provide the constructor an appropriate set of arguments. Also, the reconstruction of the iso-surface requires a generation of an entirely new `isoSurface` class object. For this example both facts do not cause any problems, since the application loops over time step directories and reconstructs a new bubble surface mesh each time in order to compute its new position.

As documented in the header section of the `isoSurface.H` file, there are some open issues involved in the iso-surface reconstruction algorithm, but they are ignored in this example. Of course, if you manage to find a way to improve the iso-surface algorithm, feel free to extend the class and share your findings with the community using your own repository as described in chapter 6.



The argument list of the `isoSurface` constructor requires two fields:

- the cell centered field (`cellIsoVals`),
- the mesh point field (`pointIsoVals`).

Since the cell centered fields are dependent variables of the algorithms in OpenFOAM, the point field needs to be computed from the cell centered field in our application.

At this point we can summarize the following characteristics about the iso-surface approach to computing bubble velocity and area:

- `isoSurface` constructor requires computation of an additional point field,
- `isoSurface` class doesn't implement the bubble area calculation,
- `isoSurface` class doesn't implement the bubble velocity calculation.

This brings us to the conclusion that additional data and functionality is required compared to what is currently available in the `isoSurface` class. Implementing these kind of calculations into global functions and data is possible, but this results in stepping away from OOD. As a result, it would be impossible to re-use the calculations on multiple bubbles easily, and re-using them outside the scope of the application where the global variables are defined would be impossible.

Therefore, the iso-surface calculation is to be encapsulated into a new class named `isoBubble` and compiled into a dynamically loadable shared library `bubbleCalc`. This way, multiple bubble objects can be instantiated easily in different applications. This also prepares the ground for possible future algorithms which operate on bubble or bubble like entities. An example of this could be computing the distance between two bubbles by computing the closest proximity of surface meshes.

The easiest way to understand how the `isoBubble` class (put simply `isoBubble`) is built would be by inspecting the class source code available in the example code repository. The interesting parts of the implementations are commented on in this section using source code listings as well as detailed descriptions of the calculations performed by the code.

Listing 44 shows the private interface of isoBubble.

Listing 44 Private isoBubble class interface.

```
class isoBubble
:
    public regIOobject
{
    // Computed point iso-field.
    scalarField isoPointField_;

    // Bubble geometry described with an iso-surface mesh.
    autoPtr<isoSurface> bubblePtr_;

    // Number of zeros that are prepended to the timeIndex()
    // for written VTK files.
    label timeIndexPad_;

    // Output format.
    word outputFormat_;

    virtual void computeIsoPointField(const volScalarField& isoField);

    fileName padWithZeros(const fileName& input) const;

public:

    // Constructors
```

Starting at the class declaration, note that the `isoBubble` inherits from the `regIOobject` class. OpenFOAM encapsulates the IO operations for objects in the `regIOobject` class, which is registered to the *object registry*. The object registry is a class that calls the write member functions of the object in question when the a writing operation is executed in the application code. It also a global object in the program that issues write requests to all objects that are subscribed to it. This kind of design represents a common OOD pattern called "Observer Pattern" (Gamma, Helm, Johnson, and Vlissides 1995), and it fits very well with CFD solvers because of the following reasons (among others):

- an object-oriented CFD application may use dozens of objects so calling write for each one causes severe code bloat,
- the objects are not necessarily written out at every time step - the write frequency follows user-defined rules that are read from a



dictionary file and applied by the registry.

Instead of having multiple write calls in the solvers (which would look something like `object.write()`) the class `Foam::Time` serves as the *Observer* or *Object Registry*, and dispatches the call to write to all subjects (registered objects).

When the time step is increased and the solution for this time step is obtained, a call to `runTime.write()` causes the `Foam::Time` registry to loop over a list of pointers to the registered objects ("Subjects" in the Observer pattern) and forwards the write call to each registered object. The actual writing will only take place, if the time step in question is actually the time step designated for the output. For example, a user can prescribe the output to take place every N time steps.

In order to write the `isoBubble` object using the write controls available in OpenFOAM (e.g. every 5 time steps, or every 0.01 seconds), the `isoBubble` class inherits publicly from the `regIOobject` class. The `padWithZeros` private member function of the `isoBubble` expands the file names so that they are made readable as a time-sequence by the `paraView` visualization application. The actual output in the specified VTK format is then delegated from the `isoBubble` to the `isoSurface` class. The files are written in an *instance* directory under the name given as a sub-argument of the `IOobject` constructor argument.

The private attributes of the `isoBubble` class are defined by the interface of the `isoSurface`. There are two very important aspects of the available `isoSurface` class constructors:

- no default constructor is available,
- the available constructor requires an additional field defined in mesh points.

The point field is not available by default in OpenFOAM simulations - OpenFOAM uses cell-centered fields as dependent variables in a mathematical model. The non-existent default constructor and a constructor that takes a field as an argument, rule out inheriting from `isoSurface`.

Since inheritance is not applicable at the first glance, we can try compo-

WARNING

Using multiple inheritance and inheriting from `isoSurface` is not possible because `isoSurface` hides the zero argument constructor from the parent class (`triSurface::triSurface()`).

sition. The `isoSurface` cannot be composed as an object, because the additional point field is needed as an argument its constructor. It would be possible to introduce a point field to correspond with the cell centered field used to reconstruct the bubble, but this would result in the user having to modify the solver code at two points each time a new bubble is reconstructed. The first point of modification would be the introduction of the point field as a global variable, and the second (expected) point of variation is the introduction of a new bubble object in the solver code. For reasons unknown to the authors, the empty constructor inherited by `isoSurface` from `triSurface` is overridden (hidden) by the available `isoSurface` constructors. As an alternative, a pointer to the `isoSurface` object can be used, which can also be zero-initialized. Using a raw pointer is not a proper way to compose `isoSurface` for a beginner OpenFOAM/C++ programmer because using raw pointers requires the programmer to be very careful about memory allocation/deallocation.

WARNING

Using raw pointers in C++ brings into question which part of the code is handling resources. In most cases, the best option is *not to use raw pointers at all*.

Note that raw pointers exist for a reason. Their existence is not wrong in itself, however, they should be used with utmost caution. For these reasons and to show how OpenFOAM smart pointers are used, the `isoSurface` object is wrapped into an OpenFOAM *smart pointer*. A smart pointer is a pointer that is easier to handle - the constructors, the assignment operator and the destructor of the smart pointer type make sure that valid memory deallocation is performed. The topic of smart pointers is covered in various C++ books and online material. The smart pointers used by OpenFOAM (`tmp`, and `autoPtr`) are covered in more detail in chapter 5.



Smart pointers are classes that encapsulate raw C++ pointers and allow for special semantics when it comes to memory operations. Some smart pointers implement reference-based object manipulation where multiple objects share only a reference to another object. This object is then destroyed when the number of references pointing to it becomes zero. Reference based manipulations allow the program to avoid creating unnecessary copies of large objects. An example of such smart pointer is the `tmp` pointer in OpenFOAM. Another use of a smart pointer is to enforce a C++ idiom called RAII, which is a complicated way of stating that the pointer is encapsulated in an object, it gets deleted by the pointer destructor as the pointer object goes out of scope. An example of such pointer is the `autoPtr` in OpenFOAM, used in the `isoBubble` to point to the `isoSurface` object. Using a smart pointer should allow the programmer to not have to worry about explicitly deallocating memory.

In our example, the smart pointer `autoPtr` causes other kinds of problems if the objects of the class in question are to be made assignable as well as copy-constructible. Because of this, the copy and assignment operations in that handle smart pointers in `isoBubble` need to be carefully programmed. If care is not taken here, accidental ownership transfer may happen between objects taking part in assignment and copy construction operations.

The `autoPtr` smart pointer assumes ownership of the object it points to. This means that if we do not define an assignment operator and copy constructor to our `isoBubble` class and instead rely on the automatically generated one, the program may produce unexpected behavior. If one bubble is assigned to another, and the first bubble is then used by the following code, runtime errors will occur. The same will apply for copy-constructing two bubbles. When one bubble is assigned to or copy

Listing 45 Standard construction and assignment operations for `isoBubble`.

```
isoBubble one(...arguments...);
isoBubble copy(one);
isoBubble assigned (...arguments...);
assigned = one;
```

constructed from another bubble such as in listing45, the new objects will

take ownership of the `isoSurface` object. As a result, the pointer to this object will be set to 0 for the object one, resulting in undefined behavior when we try to use assignment in the line `assigned = one`. To prevent this, we have forced *deep copy* operations whenever the `autoPtr` is initialized or assigned to in the `isoBubble` class, which allows copy construction and operation as one would expect: each bubble holds its own data.

Listing 46 Delegating the bubble construction to the `isoSurface` class

```
void isoBubble::reconstruct
(
    const volScalarField& isoField,
    scalar isoValue,
    bool regularize
)
{
    computeIsoPointField(isoField);

    bubblePtr_ = autoPtr<isoSurface>
    (
        new isoSurface
        (
            isoField,
            isoPointField_,
            isoValue,
            regularize
        )
    );
}
```

EXERCISE

Model the `isoBubble` class so that it does not rely on using smart pointers for the bubble mesh. Hints: Inherit from `triSurface` and implement the reconstruction member function. Is it safe to assign an `isoSurface` to `triSurface`? How is the design different than the example forcing the deep copy?

Now that we have initialized our `isoBubble` object and we are ready to copy-construct and assign `isoBubble` objects, all that is left is to con-



struct the triangulated iso-surface from a provided cell centered `isoField`. This is done in the `isoBubble::reconstruct` member function shown in listing 46, which uses `volPointInterpolation` to compute the iso-field values stored at the mesh points by inverse distance interpolation from the cell centers. The calculation of the point iso-field is refactored into a virtual member function `isoBubble::computeIsoPointField()`, because there may be alternative ways to compute the point iso-field values.

EXERCISE

Extend the `isoBubble` into a new class and provide an alternative to the `volPointInterpolation` by re-implementing the `isoBubble::computeIsoPointField` virtual member function. Find information and investigate the Template Method design pattern (Gamma, Helm, Johnson, and Vlissides 1995). Enable the RTS for the new class. Compare the results with the original `isoBubble`.

With the computed `isoPointField` and all the necessary arguments passed to the constructor, the `isoSurface` class takes care of the actual reconstruction of the iso-surface mesh. Once the iso-surface is reconstructed, we can use its public interface to perform the center and area calculations for our bubble. We compute the area of the bubble as the sum of the magnitudes of the iso-surface face area normal vectors:

$$A_b = \sum_f \|\mathbf{S}_f\|, \quad (8.1)$$

where \mathbf{S}_f is the area normal vector of the iso-surface mesh face. The bubble center is the algebraic average of all the iso-surface mesh points:

$$\mathbf{C}_b = \frac{1}{N} \sum_N \mathbf{x}_N, \quad (8.2)$$

where N is the number of points of the iso-surface mesh. The computations are performed by the `isoBubble::area` and `isoBubble::center` member functions shown in listing 47. Having re-used the `isoSurface` class and encapsulating our IO operations, as well as the computation of the bubble area and center, we can now easily write applications and instantiate as many bubble objects as we like. The `isoBubble` class is

Listing 47 Iso-bubble area and center point calculation.

```
scalar isoBubble::area() const
{
    scalar area = 0;

    const pointField& points = bubblePtr_->points();
    const List<labelledTri>& faces = bubblePtr_->localFaces();

    forAll (faces, I)
    {
        area += mag(faces[I].normal(points));
    }

    return area;
}

vector isoBubble::centre() const
{
    const pointField& points = bubblePtr_->points();

    vector bubbleCentre (0,0,0);

    forAll(points, I)
    {
        bubbleCentre += points[I];
    }

    return bubbleCentre / bubblePtr_->nPoints();
}
```

prepared for alternative computations, and can be extended with other bubble-related calculations.

In order to easily share the implementation of the `isoBubble` with others (people or client programs), the `isoBubble` implementation is set to be a part of a library, namely the `bubbleCalc` library. Listing the `$PRIMER_EXAMPLES_SRC/bubbleCalc` directory shows the library files:

```
?> ls
isoBubble.C isoBubble.dep isoBubble.H lnInclude Make
```

The library build specifications are given in the `Make` folder, in files `op-`



tions and files. The file `files` lists the files that are compiled into

Listing 48 File `files` of the `bubbleCalc` library.

```
isoBubble.C
```

```
LIB = $(FOAM_USER_LIBBIN)/libbubbleCalc
```

Listing 49 File `options` of the `bubbleCalc` library.

```
EXE_INC = \
  -I$(LIB_SRC)/finiteVolume/lnInclude \
  -I$(LIB_SRC)/meshTools/lnInclude \
  -I$(LIB_SRC)/triSurface/lnInclude \
  -I$(LIB_SRC)/surfMesh/lnInclude \
  -I$(LIB_SRC)/surfMesh/MeshedSurfaceProxy/ \
  -I$(LIB_SRC)/sampling/lnInclude

EXE_LIBS = \
  -lmeshTools \
  -lfiniteVolume \
  -ltriSurface \
  -lsampling
```

the library object code. The linker later uses this to bind to the function calls declared in the included header files and called in our application code. This file also specifies where the compiled library will be installed, in this case, and as recommended for user-defined libraries, the library is installed in the folder `$FOAM_USER_LIBBIN` to avoid polluting the library install directory of the OpenFOAM system.

The `options` file lists all the necessary paths to directories that hold the included header files used in `isoBubble`, as well as paths to the directories where the library code is installed. The pre-compiled library code allows us to extend existing code without always having to re-compile everything. The library is compiled with the OpenFOAM build system `wmake`, by issuing the command `wmake libso` within the `bubbleCalc` directory.

With the compiled library, the `isoBubble` class can be used in the example post-processing utility `isoSurfaceBubbleCalculator`. To continue examining the parts of the application that are using the `isoBubble`

implementation, open the application source file `isoSurfaceBubble-Calculator.C` in a text editor. Listing 50 shows how the object of the

Listing 50 Constructing the iso-bubble.

```
isoBubble bubble
(
    IOobject
    (
        "bubble",
        "bubble",
        runTime,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    inputField
);
```

`isoBubble` class is initialized. Inheriting from the `regIOobject` forces the client code to initialize the `isoBubble` object with an `IOobject`. The parameters of the `IOobject` constructor are defined as follows:

- "bubble" - name of the object (in our case, the name of the file we are using to save the iso-surface data),
- "bubble" - instance, or the directory where the object is stored (in this example same as the name of the file),
- `runTime` - object registry that the object registers to - we want the IO operations for the `isoBubble` to be regulated by the simulation time,
- `IOobject::` - tokens (public enumerations) that define the runtime write/read operations for the `IOobject`,
- `inputField` - tracked iso-field.

The `timeSelector` class is then used to find the time step (iteration) directories among the files and folders in the simulation case directories as shown in listing 51. Within each iteration, the iso-field is read, the `isoBubble` object is reconstructed, the area and the center of the bubble are computed, and the bubble iso-surface mesh is written to the disk. This process is illustrated in listing 52.

The post-processing application code is clean of global data and the use of



Listing 51 Initialization of time directories using the time selector and looping over them.

```

Foam::instantList timeDirs = Foam::timeSelector::select0(
    runTime,
    args
);

forAll(timeDirs, timeI)
{
    runTime.setTime(timeDirs[timeI], timeI);
}

```

Listing 52 Calculation of the bubble parameters within the time step loop.

```

inputField = volScalarField
(
    IOobject
    (
        fieldName,
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    ),
    mesh
);

bubble.reconstruct(isoField);

Info << "Bubble area = "
    << bubble.area() << endl;

Info << "Bubble centre = "
    << bubble.centre() << endl;

bubble.write();

```

the class is simplified to the class client, which means that the isoBubble is behaving in the same manner of any other OpenFOAM type. Executing the isoSurfaceBubbleCalculator in the rising-bubble-2D example case results in a series of "vtk" files stored in the bubble directory. These files are then easily viewed as a file sequence using the ParaView visualization application.

Visualization of the iso-surface shows some issues with the 2D algorithm, which are also noted in the isoSurface.H file, as shown in figure 8.2 for the final time step of the rising-bubble-2D example case. There

is no visible difference between figures 8.2a and 8.2b, however some instabilities do occur for the 2D iso-surface calculation in OpenFOAM, and they are visible in figure 8.2c.

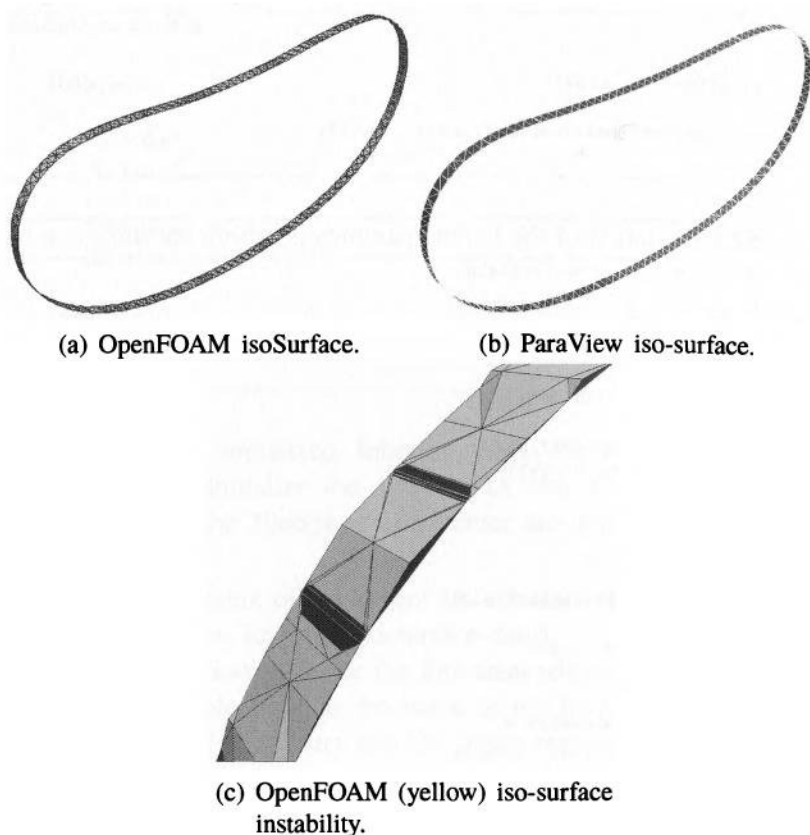


Figure 8.2: Visualization and comparison of the iso-surfaces generated by OpenFOAM and computed within ParaView.

When executing the `isoSurfaceCalculator` application, the output shown in listing 53 is produced. The output of the `isoSurfaceBubbleCalculator` can be readily processed by a script in order to compute additional quantities such as bubble velocity. This computation is done by a BASH shell script `extractIsoBubbleData`, which produces a `.dat` file that is later processed within the `ipython` console. The `extractIsoBubbleData` data extraction script parses the output of the post-processing application and stores it in a file `isoBubble.dat`, as shown in listing 54.

Listing 53 Output of the `isoSurfaceCalculator` application.

```

Time = 0
Bubble area = 7.74259
Bubble centre = (1.00217 0.998911 0.00665)
Time = 0.1
Bubble area = 12.9308
Bubble centre = (1 1.00216 0.00665)
Time = 0.2
Bubble area = 13.0444
Bubble centre = (0.999999 1.01213 0.00665)
Time = 0.3
Bubble area = 12.1002
Bubble centre = (0.999997 1.02869 0.00665)

```

The script uses standard Linux commands: `awk` for parsing regular ex-

Listing 54 Bubble data output extraction script.

```

#!/usr/bin/bash

awk '/Time = / {print $3}' $1 > time.dat
awk '/Bubble area = / {print $4}' $1 > isoBubbleArea.dat
awk '/Bubble centre = / {print $4,$5,$6}' $1 > isoBubbleCentre.dat

sed -i 's/(//g' isoBubbleCentre.dat
sed -i 's/)//g' isoBubbleCentre.dat

paste time.dat isoBubbleArea.dat isoBubbleCentre.dat > isoBubble.dat

rm -rf isoBubbleArea.dat
rm -rf isoBubbleCentre.dat

```

pressions, `sed` for inplace substitution (removal) of rounded brackets, and `paste` for redirecting the column data in a new file. This is the simplest form of a BASH shell script that analyzes OpenFOAM output. Additional simple and advanced parsers are available in the PyFoam package. In general, when writing your own post-processing applications, you can modify the application output format to fit the proper form of post-processing. Raw column data, for example, may be well handled with `xmgrace` or `gnuplot`. You may also extract data using standard Linux utilities or PyFoam.

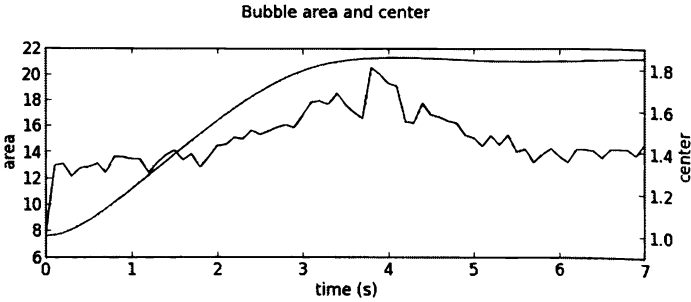


Figure 8.3: Diagram of the isoBubble data.

To visualize the evolution of the bubble area and centre in time, we execute the python script `plotIsoBubble` on the `isoBubble.dat` file:

```
?> plotIsoBubble isoBubble.dat
```

This will render the .png diagram shown in figure 8.3. The data visualization script is implemented as shown in listing 55 - the comments should make the code self-explanatory.

EXERCISE

Consider implementing the velocity calculation as a member function of the `isoBubble` class. For computing the velocity with a first order Euler temporal discretization (Chapter 1), the `isoBubble` needs to store at least one bubble center position vector from the old time step.

EXERCISE

Compute the rising velocity with a more accurate time differencing scheme. Hint: read about the *Composite Pattern* (recursive composition) of the Object Oriented Design and investigate the OpenFOAM `GeometricField` class template to find out how the OpenFOAM fields store their old values used by the temporal derivatives.



Listing 55 Bubble data visualization Python script.

```
#!/usr/bin/python

# Import system module for reading
# script arguments
import sys
# Import numerical python module for
# reading in data
import numpy as np
# Import matplotlib plotting module
import matplotlib.pyplot as plt

# Set the name of the file to the
# first script argument.
dataFile = sys.argv[1]
# Read in text data.
data = np.loadtxt(dataFile)

# Initialize the figure.
fig = plt.figure()

# Create and plot the first axis.
ax1 = fig.add_subplot(2,1,1)
ax1.set_xlabel('time (s)')
ax1.set_ylabel('area', color='b')
ax1.plot(data[:,0], data[:,1], color='b')

# Clone the axis.
ax2 = ax1.twinx()
# Plot y centre position only:
# the bubble rise is in 2D.
ax2.set_ylabel('center', color='r')
ax2.plot(data[:,0], data[:,3], color='r')

# Set plot title.
fig.suptitle('Bubble area and center')
# Save the .png image and make sure it is trimmed.
plt.savefig("isoBubble.png", bbox_inches='tight')
```

EXERCISE

Write another post-processing application that computes the bubble velocity using the α_{p1} and U fields. What are the differences in the calculated velocity and where do these differences originate from?

Further reading

Gamma, Erich et al. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2.



9

Solver Customization

9.1 Solver Design

A solver application (solver) is not structurally different from any other OpenFOAM application in the sense that it is built from functionalities implemented in different libraries. The algorithms and data structures of a numerical method are used by the solver to approximate the solution of a mathematical model that describes a physical process. A brief overview of the FVM supported by OpenFOAM is presented in chapter 1. Depending on which physical process is simulated, OpenFOAM solvers are categorized into different categories. They can be located in the OpenFOAM installation either by using a predefined alias command

```
?> sol
```

which switches to the parent directory of all the solvers available in OpenFOAM, or by changing manually to the `$FOAM_SOLVERS` environmental shell variable:

```
?> cd $FOAM_SOLVERS
```

As an example solver, we have chosen the `interFoam` solver used often for DNS of two-phase flows. The `interFoam` solver implements a mathematical model that describes the flow of two separated immiscible

fluids (phases) as a flow of a single fluid (continuum). The separation of phases is done by using an additional scalar field - the volume fraction field (α). This method is referred to as the Volume-of-Fluid (VoF) method and α denotes the filling level of the cell with the first phase and takes on the values from the interval $[0, 1]$. For more information on the VoF method, please refer to specific textbooks such as the one by Tryggvason, Scardovelli, and Zaleski 2011.

The source code files of `interFoam` are located in `$FOAM_SOLVERS/multiphase/interFoam`. Listing the contents of the `interFoam` directory, results in the following contents:

```
?> ls $FOAM_SOLVERS/multiphase/interFoam
Allwclean          interFoam.dep
Allwmake           interMixingFoam
alphaCourantNo.H   LTSInterFoam
alphaEqn.H         Make
alphaEqnSubCycle.H MRFInterFoam
correctPhi.H       pEqn.H
createFields.H     porousInterFoam
interDyMFoam       setDeltaT.H
interFoam.C        UEqn.H
```

As it can be seen from the directory listing above, there are several derivatives of the standard `interFoam`. The solvers `interDyMFoam` and `LTSInterFoam` are two of them. The first one extends `interFoam` with the capability to use dynamic meshes, whereas the `LTSInterFoam` is a steady state version of `interFoam` and employs local time stepping.

The files containing the "Eqn" suffix contain source code that define the equations of a mathematical model that the solver is implementing. This implementation is done on a very high level of abstraction, which compares well with the common human readable formulations. Obviously, `interFoam` uses the momentum equation (`UEqn.H`), the pressure equation (`pEqn.H`), the volume fraction equation (`alphaEqn.H`) as well as other supporting parts of the solution algorithm. Those supporting parts are implemented in different libraries and are placed in various subdirectories of `$FOAM_SRC`, in order to make them available to any solver.

Source code that operates on global field variables and is not implemented in a form of a function or a class, is distributed in files that are included by the solver applications. Including the file `CourantNo.H` inside solver



TIP

Whenever a source file is encountered with an *Eqn* suffix, that file contains an implementation of an equation of a mathematical model.

code, includes the calculation of the current Courant number in the solver application which is based on the field variables available at the global scope. In order to prevent code duplication, the number of such files that share common source code snippets is reduced to a minimum with the help of the build system. All shared included files are stored in specific folders such as:

```
?> ls $WM_PROJECT_DIR/src/finiteVolume/cfdTools/general/include
checkPatchFieldTypes.H fvCFD.H initContinuityErrs.H
readGravitationalAcceleration.H readPISOControls.H
readTimeControls.H setDeltaT.H setInitialDeltaT.H
volContinuity.H
```

and links to those files are linked into the specific `lnInclude` folder. The `lnInclude` folder is searched by the build system for files to be included by the solver application. Of course, the directory must be specified in the `Make/options` file of the solver application. By default, all OpenFOAM applications include the `finiteVolume/lnInclude` as the search directory for the compiler:

```
EXE_INC = \
-I$(LIB_SRC)/finiteVolume/lnInclude \
```

and this directory contains the symbolic links to all header files in the `finiteVolume` directory and its sub-directories. Therefore, the included header files shared between different applications from `cfdTools` are by default made available to any OpenFOAM application.

Other globally available functionalities that are implemented in terms of header files are listed above, and they include the following:

- time step adjustment based on the CFL condition,
- correction for volume conservation,
- determination of continuity errors,
- reading the gravitational acceleration vector,

- reading the controls of the Pressure-Implicit with Splitting of Operators (PISO) algorithm, etc.

One could argue that the calculations implemented in terms of files that are included into the application (solver) code is not a pure object-oriented approach to software design. This is true, however, the very nature of the CFD simulation is procedural: process simulation input to compute the approximative solution and store the output. Therefore, some variables have been made global and thus allways accessible to the solution process, such as the fields and the mesh. In that case, encapsulating operations that are defined in the included files into classes and imposing hierarchy on them is still possible. For example, there is nothing preventing the user to implement a hierarchy of classes that implement different strategies to modify the time step based on the Courant number. Actually, calculations that are performed using the global field variables as arguments can be encapsulated very nicely into function objects, as covered in chapter 12. However, such encapsulation is not allways necessary, and in those situations, the included source files are used.

The high level of abstraction of the OpenFOAM DSL developed for mathematical models is observed in construction of the momentum equation as shown in listing 56. The discrete operators used by the unstructured FVM

Listing 56 Assembly of the momentum equation using DSL / equation mimicing.

```
fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
    + fvm::div(rhoPhi, U)
    + turbulence->divDevRhoReff(rho, U)
);
```

(divergence `div`, gradient `grad`, curl `curl` and the temporal derivative `ddt`) are implemented as function templates in the C++ language. Using the generic programming allows the programmer to retain the same function names (that are human readable) for functions taking completely different formal parameters. For this reason, there exists a single implementation of the gradient operator and not separate implementations for tensors of different ranks: scalar, vector, symmetric tensor, and the like.



OOD is used to encapsulate things such as interpolations required by the discrete operators, and keeps their interaction invisible to the user of OpenFOAM except as a named entry in the configuration files. As already pointed out in the previous chapters, the `fvSolution` and `fvSchemes` dictionaries are responsible for the control of the numerical schemes and solvers. The equations of the mathematical model invoke discrete operators on fields such as the velocity, pressure, momentum and so forth. This is why different solvers need different entries in both the `fvSolution` and `fvSchemes` dictionaries, if the default entry is set to `none`. Hence neither the user nor the developer of a solver have to care about how any of the schemes are implemented, OpenFOAM selects the chosen schemes automatically based on the specification in the configuration file. As the RTS is enabled for the schemes, the solver must not be recompiled if a discretization changes.

TIP

OpenFOAM uses generic programming to implement the discrete differential operators. The operators dispatch the discretization and interpolation operations to a generic hierarchy of discretization and interpolation schemes, respectively. The interpolation/discretization class templates are instantiated and have been made runtime selectable. As a consequence, no change to the solver code is required when a different scheme is chosen.

9.1.1 Fields

As described in chapter 1, the unstructured FVM requires a discrete subdivision of the flow domain into a mesh of finite volumes, onto which the physical fields (e.g. pressure, velocity, temperature) are mapped. The fields, together with the mesh and the solution control (implemented by the `Time` class), are initialized by the solver at the beginning of the simulation but before the time loop starts. The initialization part of the solver code can be inspected in the main solver application file `interFoam.C` is shown in listing 57. Most of the above listed included (header) files are made globally available. Some header files are solver-specific and in that case they are not available to any application, but are stored in the solver source directory. Examples of solver-specific files are `createFields.H` and `readTimeControls.H`, which read the solver

Listing 57 A typical set of include files found in the start of OpenFOAM solvers

```
#include "setRootCase.H"
#include "createTime.H"
#include "createMesh.H"

pimpleControl pimple(mesh);

#include "initContinuityErrs.H"
#include "createFields.H"
#include "readTimeControls.H"
#include "correctPhi.H"
#include "CourantNo.H"
#include "setInitialDeltaT.H"
```

specific fields and solver related control structures. Whichever fields are initialized in the `createFields.H` will be expected by the solver to be present in the initial (0) directory of the simulation case when the solver is invoked.

9.1.2 Solution Algorithm

The momentum conservation equation contains the term on the right hand side that models the volumetric force density of the pressure acting on the fluid. The pressure field is not known at the beginning of the simulation. The pressure acting upon the fluid enforces a change in the momentum, however, the mass conservation (continuity) equation needs to be satisfied at all times. For an incompressible flow, this condition causes a tight coupling between the continuity equation and the momentum equation. This equation coupling is usually referred to as *pressure-velocity coupling* and there are a multitude of algorithms developed in CFD whose sole purpose is handling the equation coupling while keeping the solution process stable.

The coupling might be addressed by a simultaneous solution of the algebraic equation system using *block-coupled* solvers. A starting point in the research of the block coupled solution algorithms on unstructured meshes in general would be the work by Darwish, Sraj, and Moukalled 2009.



Developments targeting specifically OpenFOAM are those of Clifford and H Jasak 2009 and Kissling, Springer, Hrvoje Jasak, Schütz, Urban, and Piesche 2010, among others. Although it is intuitive to the strong nature of the equation coupling, this approach has historically not been used because of the prohibitive memory constraints of the computers at the time when the pressure-velocity coupling problem has been encountered in CFD. As a consequence, *segregated algorithms* have been developed that modify the original set of equations into a modified set that allows a separate solution of each equation.

Detailed information on CFD solution algorithms for pressure-velocity coupling can be found in textbooks on CFD like the book by Ferziger and Perić 2002 and Patankar 1980. OpenFOAM implements two major pressure-velocity coupling algorithms: the PISO algorithm originally developed by Issa 1986 and the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm, originally developed by S.V. Patankar and D.B. Spalding 1972.

Starting from OpenFOAM versions of 2.0 or newer the user interface to the pressure-velocity coupling algorithms has experienced a major refactoring which consolidates both existing algorithms into a single one named PIMPLE. Instead of reading the respective settings from `fvSolution` in each solver explicitly, a new class named `pimpleControl` takes care of all of this on it's construction. This lead to a simplification of `readControls.H` for each solver. Please note that this solely affects the way the particular parameters are read from `fvSolution` and not the implementation of the algorithm in the solver itself. One can use `pimpleControl` and still implement an entirely different pressure-velocity coupling algorithm instead.

9.2 Customizing the Solver

A majority of solver customization involves modifying the mathematical model equations in one way or the other. Entirely new model equations may be introduced, equation terms may be removed as their influence on the solution is neglected, and new terms can be added to account e.g. for new forces acting on the fluid. While it is impossible to cover all of the possible complexities and difficulties that can arise during the

customization of solvers, there are a few examples that are covered in this chapter that should help in understanding problems related to solver customization and how to overcome them. One of the first modifications to a solver is typically adding new fields or material properties to work with. Regardless of the purpose, all of those things have to be read by the solver application from the simulation case. There are various files available in the simulation case, where different types of data are read from. Lets start with one of the more basic operations involved: looking up a value in a dictionary. OpenFOAM configuration (dictionary) files and the data structure they are based on are both described in detail in chapter 5. Still, basic information in working with dictionaries is provided also in this chapter, to make the solver customization description more self sustained.

9.2.1 Working with Dictionaries

OpenFOAM configuration files are called *dictionaries* (dictionary files) and are used to provide configuration parameters for the solver application. Between the various dictionaries such as `transportProperties`, `controlDict`, or `fvSolution`, the user has complete control over solvers, material properties, time stepping, and the like. In this section, accessing an existing and a new dictionary as well as looking up a value and loading it into solver scope is covered. Examining how the `icoFoam` solver looks up the material properties from the `constant/transportProperties` dictionary will serve as a simple example of how to dictionaries are handled. To begin, open the `createFields.H` file located at `$FOAM_APP/solvers/incompressible/icoFoam/createFields.H`. At the beginning of this header file, there is an instantiation of the `IObject` dictionary object as shown in listing 58. The `IObject` declaration is commented out in order to indicate the purpose of each argument. More details on these arguments are provided later in this section. The code of listing 58 initializes the dictionary as a global variable, so retrieving values from it remains straightforward. The next few lines of the `createFields.H` file to contain the code that initializes the viscosity (`nu`) with a value contained in the `transportProperties` the dictionary:

```
dimensionedScalar nu
(
    transportProperties.lookup("nu")
);
```



Listing 58 Construction of an IOdictionary

```

Info<< "Reading transportProperties\n" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        // Name of the file
        "transportProperties",
        // File location in the case directory
        runTime.constant(),
        // Object registry to to which the dict is registered
        mesh,
        // Read strategy flag: read if the file is modified
        IOobject::MUST_READ_IF_MODIFIED,
        // Write strategy flag: do not re-write the file
        IOobject::NO_WRITE
    )
);

```

To this purpose, the IOdictionary class has a function called `lookup(word& w)` where `word` is simply the string representing the name of the looked up variable. There are other lookup methods as well, such as `lookupOrDefault<T>(word& w, T default)`, which tries to find the variable or property with name `w` in the particular dictionary and lookup its value. If the variable doesn't exist, a default value is used. The example `transportProperties` entry (listing 59) shows how the `nu` entry is defined, together the appropriate dimensions and scalar value. The physical values that OpenFOAM relies on are dimensioned. More information on how the dimension system in OpenFOAM works can be found in chapter 5. It is essential to assign the above shown entry for `nu` to a `dimensionedScalar`.

Listing 59 Example definition of `nu` in `transportProperties`

```

nu          nu [ 0 2 -1 0 0 0 0 ] 0.01;

```

`dimensionedScalar`. Using a scalar instead would compile as well, but the solver would crash when trying to look up `nu` and assigning it to a `dimensionedScalar`.

The following example shows how transport properties are looked up by the `interFoam` solver, which is a bit more complicated. The important source code is located at `$FOAM_APP/solvers/multiphase/inter-`

Foam/ At the beginning of the `createFields.H` file, the `twoPhaseMixture` class is instantiated (see listing 60). There are two member func-

Listing 60 Instantiation of `twoPhaseMixture`

```
Info<< "Reading transportProperties\n" << endl;
twoPhaseMixture twoPhaseProperties(U, phi);

volScalarField& alpha1(twoPhaseProperties.alpha1());

const dimensionedScalar& rho1 = twoPhaseProperties.rho1();
const dimensionedScalar& rho2 = twoPhaseProperties.rho2();
```

tions (`rho1()`, `rho2()`), used to lookup the material properties of the `twoPhaseMixture` class. Examining the class source code is necessary, in order to find the code that performs the actual dictionary access. The source code for this class is located at `$FOAM_SRC/transportModels/incompressible/incompressibleTwoPhaseMixture/twoPhaseMixture.C`.

EXERCISE

Search the way up from the class constructor to the lines where material properties for each phase are looked up and returned by the public member functions `rho1()` and `rho2()` that are called in the `createFields.H` header.

9.2.2 The Object Registry and `regIOobjects`

As the fields and the mesh are used in the form of global variables in the OpenFOAM solver applications, tracking all of them and dispatching calls to their member functions explicitly would involve a lot of code repetition, which is bad. An example of such a clustered call dispatch is a request of the solver to write out all fields to the hard disk. In the case of using objects directly, the names of the objects would be hardcoded and changing the names would introduce a cascade of changes in the application code. Also, the code that implements such calls would need to be copied on multiple places. For example, the same code responsible for the field output would then be copied to every application that relies on the same set of fields with the same variable names. Using included



header files as covered in section 9.1 would be possible for the solver family that relies on same field variables. But changing a single field variable would render such a header file unusable for the entire solver family. This approach hence represents a *stiff* software design, or a software design which *does not scale well*. A stiff or not-scaling software design is a design for which a single extension requires the modification of existing code, and furthermore, the modification occurs often at multiple places in the existing code base.

Because of this issue, the logic behind coordination of operations for multiple objects has been encapsulated into a class, that can then be re-used at many places in the OpenFOAM code. To this purpose, an *object registry* has been implemented: an object that registers other objects to itself and then dispatches (forwards) the call to its member functions to the registered objects. The object registry is an implementation of the *Observer Pattern* from the OOD and it is described in more detail in section 8.3. Additionally, there is a great review of the object registry as well as the registered object classes on the OpenFOAM Wiki page ¹.

An example of the use of an object registry is a boundary condition implementation where the boundary condition that operates on one field, requires access to another field. The total pressure boundary condition (`totalPressureFvPatchScalarField`) is such a boundary condition that requires access to multiple fields in order to update the field it is assigned to.

EXERCISE

Find out what is the `totalPressureFvPatchScalarField` boundary condition meant to be used for. Which member function performs the actual calculations? How are the alternative calculations implemented? Can you think of an alternative runtime selectable implementation for the alternative calculations?

The total pressure boundary condition updates the field it has been assigned to as it is done by all other boundary conditions in OpenFOAM, using the `updateCoeffs` member function as shown in listing 61. However, the `updateCoeffs()` dispatches the calculation to the overloaded

¹http://openfoamwiki.net/index.php/Snip_objectRegistry

Listing 61 Total pressure boundary condition dispatch to object registry.

```
void Foam::totalPressureFvPatchScalarField::updateCoeffs()
{
    updateCoeffs
    (
        p0(),
        patch().lookupPatchField<volVectorField, vector>(UName())
    );
}
```

updateCoeffs. The second argument of the updateCoeffs call shown in listing 61 accesses the fvPatch constant reference attribute of the boundary field using the patch() member function. On the other hand, the fvPatch class stores a constant reference to the finite volume mesh, which inherits from objectRegistry and is therefore *an* object registry. The lookupPatchField is defined as a template member function of the fvPatch class template, in the file fvPatchFvMeshTemplates.H and it is shown in listing 62. The return type declaration of the function is

Listing 62 Looking up a geometric internal field from within a boundary mesh patch.

```
template<class GeometricField, class Type>
const typename GeometricField::PatchFieldType&
Foam::fvPatch::lookupPatchField
(
    const word& name,
    const GeometricField*,
    const Type*
) const
{
    return patchField<GeometricField, Type>
    (
        boundaryMesh().mesh().objectRegistry::template
        lookupObject<GeometricField>(name)
    );
}
```

dependant on the template parameter GeometricField and therefore requires the typename keyword, in order to make the compiler aware that the PatchFieldType is indeed a type. The more important part of the member function template is the return statement that obviously makes use of the object registry functionality, which is inherited to the fvMesh from the object registry class *objectRegistry*. Since fvPatch is a class



template, the call to the base class member function is a bit convoluted. `lookupObject` is a member function template of the `objectRegistry` base class, and this must be specified at the member function call site using the `template` keyword. Looking past all the C++ template code, the finite volume mesh boundary patch accesses the entire boundary mesh, then the corresponding volume mesh and asks the volume mesh to look up a field with a specific name (`name`). This call path through the involved classes as described for this example allows the total pressure boundary condition to access a field from a mesh, based on the field name parameter.

9.3 Implementing a new PDE

In this section a new solver is implemented by adding a new PDE to the `interFoam` solver for simulating two incompressible immiscible fluid phases. The purpose of the PDE and its supporting code to show how the implementation is performed, and not to model a realistic physical transport process. Modeling of transport phenomena on and across fluid interfaces requires careful and rigorous derivation. For the most part, the equation stems from an implementation available in the `compressibleInterFoam` solver. There, the heat transfer across the interface is accounted for, due to the use of equations of state which couple pressure, temperature and density. For the sake of simplicity, thermodynamic equations of state are omitted and laminar flow is assumed.

9.3.1 Additional Model Equation

As an example of a model of the unsteady passive scalar transport in a laminar two phase flow, the following equation is used

$$\frac{\partial \rho T}{\partial t} + \nabla \cdot (\rho \mathbf{U} T) - \nabla \cdot (D_{eff} \nabla T) = 0, \quad (9.1)$$

where T denotes the temperature, ρ is the density, \mathbf{U} is the velocity, D_{eff} is the thermal conductivity coefficient of the phase mixture.

The `interFoam` solver uses a Navier-Stokes equation model in a single field formulation to model the two-phase flow of two immiscible incompressible fluid phases. This model describes the flow of two immiscible

WARNING

The proposed equation is used to show how to *implement* a mathematical model modification to a solver in OpenFOAM by adding a new model equation. Careful mathematical modeling is required to derive proper model equations for real-world problems.

fluid phases as the flow of a single continuum, by introducing an additional scalar (volume fraction field) to distinguish between fluid phases. More information on two-phase flow modeling can be found in the book by Tryggvason, Scardovelli, and Zaleski 2011, and following the references found therein. There are many other textbooks and scientific publications available, that describe the mathematical modeling of two-phase flows and the reader is directed there to find more details on the VoF method as well as the mathematical modeling of two-phase flows.

Any cells with an α value in between with be considered interface cells with material properties weighted as a mixture of the two primary phases, since α is defined as the volume fraction by

$$\alpha = \frac{V_1}{V}, \quad (9.2)$$

where V_1 is the volume occupied by phase 1 in the cell that has the total volume V . The properties of the single continuum are then modeled using the volume fraction field, forming mixture quantities, such as the mixture viscosity

$$\nu = \nu_1 \alpha_1 + (1 - \alpha_1) \nu_2 \quad (9.3)$$

or mixture density

$$\rho = \rho_1 \alpha_1 + (1 - \alpha_1) \rho_2, \quad (9.4)$$

where ν_1 , ν_2 and ρ_1 , ρ_2 are the kinematic viscosities and densities of respective two fluid phases. The effective heat conduction coefficient in equation 9.1 D_{eff} is modeled a similar way in this example, using the α_1 field

$$D_{eff} = \frac{\alpha k_1}{C_{v1}} + \frac{(1 - \alpha) k_2}{C_{v2}}. \quad (9.5)$$

Here, k_1 , k_2 and C_{v1} , C_{v2} represent the conduction coefficients and heat capacities of respective fluid phases.



The heat conduction coefficient modeled by equation 9.5 is based on the volume fraction field in the similar way as other phase properties. Using the volume fraction field values this way assigns for this example constant values of the physical properties to the bulk regions of the two fluid phases. The region of the fluid interface will have values of α_1 within the interval $[0, 1]$ and will therefore define a transition region between the two constant values of the respective phase properties. The shape of the profile of the transition region will be determined by the nature of the model that describes the mixture property, like the conduction coefficient (see equation 9.5). How accurately this approach can be applied on the real physical problem of heat transfer across a moving interface is not important for describing how to perform solver modifications in OpenFOAM and is therefore left out of scope.

9.3.2 Preparing the Solver for Modification

Before changing the source code of an existing solver, a new copy of the solver application must be created. The source code directory of the `interFoam` solver should be copied to the personal applications directory and renamed:

```
?> cp -r $FOAM_APP/solvers/multiphase/interFoam/ \
    $FOAM_RUN/../applications/
?> cd $FOAM_RUN/../applications/
?> mv interFoam heatTransferTwoPhaseSolver
?> cd heatTransferTwoPhaseSolver
?> mv interFoam.C heatTransferTwoPhaseSolver.C
```

The directory should be cleaned up from the variations of the `interFoam` solvers and the compilation scripts which are not used in this example:

```
?> rm -rf interDyMFoam/ LTSInterFoam/ MRInterFoam/ \
    interMixingFoam/ porousInterFoam/ Allwmake Allwclean
```

Since the solver source code files have been renamed, the `Make/files` must be modified so it reflects the changes. The `Make/files` config file should contain only the lines shown here:

```
heatTransferTwoPhaseSolver.C
EXE = $(FOAM_USER_APPBIN)/heatTransferTwoPhaseSolver
```

letting the `wmake` build system know that an executable application is to be built and installed in the user application binaries directory, from the renamed solver source code file. So, this will instruct `wmake` to compile the file `heatTransferTwoPhaseSolver.C` along with its dependencies and create an executable solver called `heatTransferTwoPhaseSolver`. Executing `wmake` at this point should result in a successful compilation of the solver under the new name, but with exactly the same implementation as the copied `interFoam` solver. After the successful compilation of the solver copy, the modifications in the application directory source code can be applied.

9.3.3 Adding new Entries into `createFields.H`

Two new required material phase properties must be looked up from the dictionary: the heat conduction coefficient k and the heat capacity C_v . The code shown in listing 64 can be inserted anywhere in the `createFields.H` file, as long as the point of insertion lies after the `twoPhaseMixture` object initialization shown in listing 63. The heat

Listing 63 Initialization of the two-phase mixture class object `twoPhaseProperties`.

```
Info<< "Reading transportProperties\n" << endl;
twoPhaseMixture twoPhaseProperties(U, phi);
```

capacity of type `dimensionedScalar` will be loaded from each phases' subdictionary in the `transportProperties` dictionary file. The dimension of the scalar is defined here as well. Heat capacities for each phase will be loaded similarly from the same dictionary, as shown in listing 65.

Now that the material properties are read from the dictionary as ready-to-use dimensioned scalar objects, the temperature field `T` is initialized as shown in listing 66. With all of the dictionary lookup and declaration complete, the new model equation 9.1 is to be added to the solution algorithm.

9.3.4 Programming the Model Equation

In order to keep the main solver code organized, the code that implements the model equations can be separated into equation files. Therefore, the



Listing 64 Looking up heat conduction coefficients from the `twoPhaseProperties` dictionary.

```

dimensionedScalar k1
(
    "k",
    dimensionSet(1, 1, -3, -1, 0),
    twoPhaseProperties.subDict
    (
        twoPhaseProperties.phase1Name()
    ).lookup("k")
);

dimensionedScalar k2
(
    "k",
    dimensionSet(1, 1, -3, -1, 0),
    twoPhaseProperties.subDict
    (
        twoPhaseProperties.phase2Name()
    ).lookup("k")
);

```

temperature equation code is placed into a separate header file `TEqn.H` and this file is then included in the solver application code. The code defined in the `TEqn.H` file implements two operations. It calculates the coefficient D_{eff} and it solve the passive temperature transport PDE given by equation 9.1.

The D_{eff} coefficient is calculated as a linear weighted average based on the cell center `alpha1` value (equation 9.5). For this example, the same approach to computing the D_{eff} coefficient has been applied as the one used in the `compressibleInterFoam` solver. The code shown in listing 67 should be added to the `TEqn.H` file and the file should be saved in the solver directory. Since the D_{eff} coefficient will vary from cell to cell, it is implemented as as a `volScalarField`, as opposed to a single scalar.

With the diffusion coefficient field setup, the transport equation can be implemented as shown in listing 68. Now that `TEqn.H` is complete, it needs to be added to the solution algorithm implemented in the main solver application file `heatTransferTwoPhaseSolver.C`. The `TEqn.H` should be inserted at a point after the momentum equation as shown in listing 69. That completes the source code modifications required to add the heat transport equation to the solution algorithm of the `interFoam` solver. The solver directory should be cleaned up from the old files

Listing 65 Looking up heat capacity coefficients from the `twoPhaseProperties` dictionary.

```
dimensionedScalar Cv1
(
    "Cv",
    dimensionSet(0, 2, -2, -1, 0),
    twoPhaseProperties.subDict
    (
        twoPhaseProperties.phase1Name()
    ).lookup("Cv")
);

dimensionedScalar Cv2
(
    "Cv",
    dimensionSet(0, 2, -2, -1, 0),
    twoPhaseProperties.subDict
    (
        twoPhaseProperties.phase2Name()
    ).lookup("Cv")
);
```

generated by the build process, and then the solver application is to be compiled:

```
?> wclean
?> wmake
```

Once the new solver is implemented a simulation case needs to be set up which is compatible with the new solver.

9.3.5 Setting up the Case

With the new solver created, new initial conditions, boundary conditions, material properties, and solver control input parameters are now required to handle the heat transfer computation. Each of these new items are covered in this section, as well as where they need to be inserted into the simulation case files. First, the rising bubble test case should be copied the code repository and renamed as a new case:

```
?> cp -r rising-bubble-2D rising-bubble-heat-transfer
?> cd rising-bubble-heat-transfer
```



Listing 66 Initialization of the temperature volScalarField.

```

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

```

Listing 67 Effective D_{eff} coefficient implementation.

```

volScalarField Deff
(
    "Deff",
    (alpha1*k1/Cv1 + (scalar(1)-alpha1)*k2/Cv2)
);

```

The `alpha1.orig` file is to be used as a starting point for the temperature field T . The `alpha1.orig` is to be copied as follows:

```
?> cp /0/alpha1.orig /0/T
```

The object name and its dimensions need to be updated in the `T` file, and the rest of the boundary can be left conditions untouched:

```

FoamFile
{
    version      2.0;
    format        ascii;
    class         volScalarField;
    location      "0";
    object        T;
}

dimensions      [0 0 0 1 0 0 0];

```

To set the initial conditions for the new temperature field, the `setFieldsDict` file in the system directory needs to be edited. The the bubble temperature is set to an initial value of 500:

Listing 68 Heat transport equation implementation.

```
solve
(
    fvm::ddt(rho, T)
  + fvm::div(rhoPhi, T)
  - fvm::laplacian(Deff, T)
);
```

Listing 69 Including TEqn.H into the solution algorithm.

```
while (pimple.loop())
{
    #include "UEqn.H"
    #include "TEqn.H" // <- Insert here

    // --- Pressure corrector loop
    while (pimple.correct())
    {
        #include "pEqn.H"
    }

    if (pimple.turbCorr())
    {
        turbulence->correct();
    }
}
```

```
sphereToCell
{
    centre (1 1 0);
    radius 0.2;
    fieldValues
    (
        volScalarFieldValue alpha1 1
        volScalarFieldValue T 500
    );
}
```

The `transportProperties` file in the `constant` directory should also be modified. The `k` and `Cv` coefficient values are added to each phase. We have chosen the values arbitrarily:

```
phase1
{
    transportModel Newtonian;
    nu          nu [ 0 2 -1 0 0 0 0 ] 1.48e-05;
    rho         rho [ 1 -3 0 0 0 0 0 ] 1;
    k           k [ 1 1 -3 -1 0 0 0 ] 1;
```



```

    Cv          Cv [ 0 2 -2 -1 0 0 0] 10;
}

phase2
{
    transportModel Newtonian;
    nu          nu [ 0 2 -1 0 0 0 0] 1e-06;
    rho         rho [ 1 -3 0 0 0 0 0] 1000;
    k           k [ 1 1 -3 -1 0 0 0] 10;
    Cv          Cv [ 0 2 -2 -1 0 0 0] 100;
}

```

Finally, the `TFinal` is added to the `fvSolution` file. Simply copying the `UFinal` configuration entry and renaming it `TFinal` is sufficient.

```

TFinal
{
    solver      PBiCG;
    preconditioner DILU;
    tolerance    1e-06;
    relTol       0;
}

```

These are the options that set the solver for the heat transfer equation, as well as its parameters. That should be the final case configuration change needed to run the new solver with an added new model equation. The solver can be run within the case directory and the results of the simulation can be analyzed.

9.3.6 Executing the solver

To execute and test the new solver, either the modified simulation case resulting from the previous section, or the prepared simulation case in the `chapter9` sub-directory of the example case repository, under the name `rising-bubble-heat-transfer`, can be used.

In order to run the case, make sure that the library and the application code of the example code repository is compiled automatically by invoking

```
?> ./Allwmake
```

in the main directory of the example code repository. To start the solver, simply execute the following command within the simulation case direc-



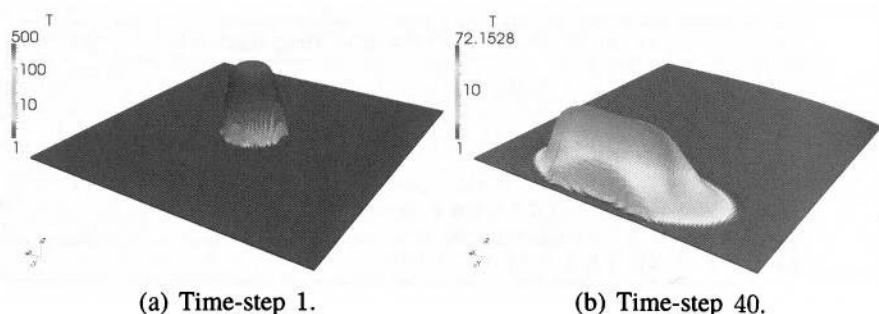


Figure 9.1: Temperature distribution for the 2D rising bubble simulation case for two chosen simulation times.

tory:

```
?> heatTransferTwoPhaseSolver
```

Figure 9.1a shows the distribution of the temperature field T at the first time step of the simulation for the two-dimensional rising bubble. The initialization is done in the same way as for the volume fraction field α_1 , using the `setFields` application configured by the `system/setFieldsDict` dictionary configuration file. The bubble temperature is set to be (much) higher than the temperature of the surrounding fluid, which should cause the bubble to release heat flux to its surroundings as it rises.

Figure 9.1b shows the temperature distribution in the time step 40, where the bubble approaches the upper wall. All wall boundary conditions for the temperature have been set to *adiabatic heat flux*: the temperature gradient is set to zero.

The images shown in Figure 9.1 have been produced by the `Warp by scalar filter`, available in the `paraView` visualization application. The temperature scale is configured to be logarithmical, and the warping is scaled by a constant small factor in order for the warped field to be visible on the image.



EXERCISE

Heat conduction in the layered cylindrical and planar geometries reduces the dimensionality of the heat transfer equation. As a result, the solution can be obtained by using Fourier series. As an exercise, program a new heat conduction solver, using the existing `heatTransferTwoPhaseSolver` solver.

Hint: find and comment out those terms of the model that model any kind of convection. Compare the temperature distribution with the exact equation solution for different mesh densities.

Further reading

- Clifford, I and H Jasak (2009). “The application of a multi-physics toolkit to spatial reactor dynamics”. In: *International Conference on Mathematics, Computational Methods and Reactor Physics*.
- Darwish, M, I Sraj, and F Moukalled (2009). “A coupled finite volume solver for the solution of incompressible flows on unstructured grids”. In: *Journal of Computational Physics* 228.1, pp. 180–201.
- Ferziger, J. H. and M. Perić (2002). *Computational Methods for Fluid Dynamics*. 3rd rev. ed. Berlin: Springer.
- Issa, R. I. (1986). “Solution of the implicitly discretised fluid flow equations by operator-splitting”. In: *Journal of Computational physics* 62.1, pp. 40–65.
- Kissling, Kathrin et al. (2010). “A coupled pressure based solution algorithm based on the volume-of-fluid approach for two or more immiscible fluids”. In: *V European Conference on Computational Fluid Dynamics, ECCOMAS CFD*.
- Patankar, Suhas (1980). *Numerical heat transfer and fluid flow*. CRC Press.
- S.V. Patankar and D.B. Spalding (1972). “A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows”. In: *International Journal of Heat and Mass Transfer* 15.10, pp. 1787–1806.
- Tryggvason, G., R. Scardovelli, and S. Zaleski (2011). *Direct Numerical Simulations of Gas-Liquid Multiphase Flows*. Cambridge University Press. ISBN: 9780521782401.



10

Boundary Conditions

For this chapter, a solid understanding of how all mesh components interact with each other to assemble the final mesh structure is required. Chapter 2 covers the details on the mesh and its structure and section 1.3 covers the numerical background of the boundary conditions in the FVM.

10.1 Numerical Background of a Boundary Condition

In the FVM one algebraic equation is generated per each cell when an implicit method is used to discretize the model equation. The coefficients and the dependent variables of the discrete algebraic equation are determined by the way the mesh topology is defined and by the values that are interpolated at the face centers. In the case when the finite volume face belongs to the domain boundary, the value is prescribed directly or indirectly by applying boundary conditions (see section 1.3). Without boundary conditions, the values on the boundaries cannot be determined and the algebraic equation system cannot be completed. The algebraic equation system remains incomplete because the boundary face values are left undetermined in the discretized equation when boundary conditions are not applied.

In the finite-volume mesh, each boundary face belongs to a boundary patch (a collection of boundary faces) and each boundary patch is defined to be of a certain type. A collection of boundary patches builds the boundary mesh. The type of the boundary patch does limit the choice of boundary conditions for all flow fields, but is not a boundary condition in itself. An overview over the different boundary types is provided in chapter 2.

Any field file of an OpenFOAM case, contains two different entries: `internalField` and `boundaryField`. The `boundaryField` describes how the values on the boundaries (boundary patch fields) are prescribed and the `internalField` does the same for the volume center values (internal field). The values on the boundary of other geometric fields (point field and surface field) are determined in a similar way. Details on boundary field operations are provided in chapter 2 as well as the *OpenFOAM User Guide* 2013.

10.2 Boundary Condition Design

Before going into details of how boundary conditions are implemented in OpenFOAM, the process of defining boundary conditions from a user's perspective is covered in this section. Additionally, how the boundary condition definition relates to the objects that describe the fields is emphasized.

10.2.1 Internal, Boundary and Geometric Fields

As stated in previous chapters, the boundary of the domain consists of different boundary-patches that are usually defined in the `constant/polyMesh/boundary` file. Each boundary-patch in turn is defined as a set of cell faces. In order to fully define a flow field that is used by a solver, both the internal field values and the values of the boundary fields, need to be defined. The initial state of the fields involved in the simulation are stored in the 0 time step directory. The boundary field values are defined via the boundary conditions that either prescribe fixed face center values or compute them from the internal field values.

As an example, consider a part of the configuration file for the dynamic



pressure field `p_rgh` of the simulation case `rising-bubble-2D`:

```
internalField  uniform 0;

GeometricBoundaryField
{
    bottom
    {
        type          zeroGradient;
    }
}
```

The `internalField` keyword relates to the values stored in the cell centers (uniform field with value 0), and the boundary condition for the bottom mesh patch is defined to be of type `zeroGradient`, which is described in detail in chapter 1.

OpenFOAM makes use of various field types: a `volVectorField` to store the velocity `U`, a `volScalarField` for the pressure field `p` and a `surfaceScalarField` to handle the flux `phi`, just to name a few. A brief look into any solver shows that there are many of operations being executed on those fields. In order to illustrate some common operations on a geometric field, the `volScalarField` `p` serves as the source of examples:

Accessing field values is simply done by passing the particular cell label to the `[]` operator of the field. In this example we choose cell 4538:

```
const label cellI(4538);
Info<< p[cellI] << endl;
```

Accessing values on the boundary is slightly more complex, because the `volScalarField` does not store any values on cell faces directly. Boundary values are determined by the boundary conditions defined on the particular boundary. Calculating the maximum value of `p` on the first boundary patch of the mesh could be achieved using the following code:

```
const label boundaryI(0);
Info<< "max(p) = "
    << max(p.GeometricBoundaryField()[boundaryI])
    << endl;
```

The values on the domain boundary are treated and accessed differently. This is due to the boundary conditions that define the values present on

WARNING

The order in which the boundary patches are stored in the boundary mesh is the same for all fields and it is determined by the way the patches are listed in the `polyMesh/boundary` file. The `boundaryI` index is fixed in the example above only because the reading of the entire mesh is not relevant for this example.

each of the domain's boundaries. By default any field returns values of its `internalField`, when accessed with operator `[]`. To access values of the boundary field the `GeometricBoundaryField()` member function must be called. This returns a list of boundary patches, one boundary patch for each mesh boundary. Each element is an abstract representation of the boundary condition chosen by the user for this patch. Depending on the type of field, this representation either inherits from `fvPatchField` or `pointPatchField`, though the first one is the one most commonly used.

In OpenFOAM there is a concept of a geometrical field, which is defined as a field that maps tensorial values to points in a geometrical mesh. The values are mapped not only to internal points of the mesh, but also to the mesh boundary points. Since different geometrical meshes are implemented as different types in OpenFOAM, a natural consequence is to model the geometrical field concept as a class template. For example, if we imagine a geometrical mesh to be consisted of line segments, and call it a *line mesh*, the corresponding model of the geometrical field concept that works with the line mesh would be named *line field*. The line field would map tensorial values to the center of each line (internal field values), and to *two boundary end points* of the line mesh (boundary field values). The class template that implements the geometrical field concept is named `GeometricalField`, and its instantiations result in different geometrical field models, that map to different kinds of meshes. In OpenFOAM different models of geometrical fields are available:

1. The first category are the well known fields like `volScalarField` and `volVectorField`, that store the data in the cell center. On the boundary, boundary conditions must be applied to mimic values stored in the face center. The suffice the naming convention of



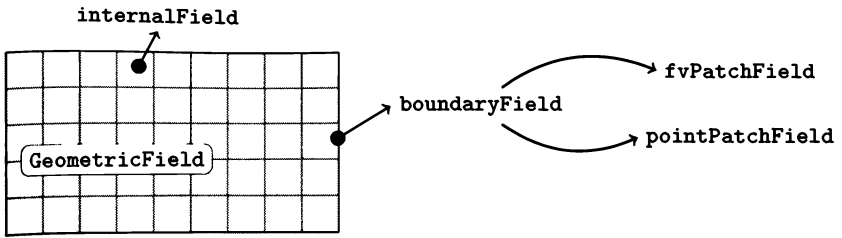


Figure 10.1: Composition of a GeometricField for a 9×5 cell mesh.

`vol*Field`.

2. The second category are fields that store data in the face center, for each face of the mesh. This is not limited to the boundary of the domain. One of these field types is the `surfaceScalarField` that is used to define the flux ϕ between two adjacent cells. All fields of this category are named `surface*Field`.
3. The third category contains the `pointScalarField` or `pointVectorField` types. Fields of this category are likely to be missed when talking about OpenFOAM fields: Fields that store the data in the points of the mesh. Each point in the mesh has its own value and on the boundaries, boundary conditions must be defined for the points on the boundary and not for the face centers. Looking for `point*Field` in the source code will show all fields of this category.

After this clarification, we can dive into the relationship between fields and boundary conditions. From the design perspective, the boundary conditions in OpenFOAM are modeled as fields that are mapped to the domain boundaries bundled together with the calculation that modifies those boundary values - the boundary *condition*. The `GeometricField` class template provides member functions which simplify the formulation of boundary conditions. Some functionalities of the member functions include computing values of the boundary fields and taking values stored in the adjacent internal cells as arguments. Both are key requirements to an implementation of any boundary condition. More information on which member functions are supposed to do which task is provided in the following section 10.2.2. The boundary fields - and thus the boundary conditions - are encapsulated together with the `internalField` to form `GeometricField` (see figure 10.1). Any of the three geometric field

models, is a typedef for a GeometricField with appropriate template arguments.

Before going into detail about the specific working principles of boundary conditions, more details on the GeometricField need to be covered. Besides viewing boundary conditions of a field stored in the 0 sub-directory of the simulation case, the described structure of boundary conditions can be also investigated in the source code. For example, consider the declaration part of the GeometricField class template shown in listing 70. From listing 70 it is obvious that the GeometricField is derived from DimensionedField. This means that any default arithmetic operations

Listing 70 GeometricField declaration.

```
template<class Type, template<class> class PatchField, class GeoMesh>
class GeometricField
:
    public DimensionedField<Type, GeoMesh>
{
```

executed on any instantiation of the GeometricField class template (a geometric field model) will be performed excluding the boundaries.

WARNING

Deriving from a class results in inheriting its member functions. In case of the GeometricField, arithmetic operators are inherited from DimensionedField. The DimensionField models the *internal field values*, since the boundary fields are *composed* by GeometricField with the GeometricBoundaryField attribute. Therefore, overloaded arithmetic operators from DimensionedField *exclude boundary field values*.

This makes sense, because the values on the boundaries should only be determined by the respective boundary conditions. Using the additional assignment operator GeometricField::operator==, the operations are extended to take the boundary fields into account as well.

Values on the boundary can be overwritten from outside of the actual boundary condition, until the boundary conditions are evaluated again by



WARNING

Operator `GeometricField::operator==` is not a logical equality comparison operator, it extends the assignment of the `GeometricField` to include boundary field values. It is *added* to the `GeometricField` interface to include arithmetic operations on the *composed* boundary fields.

the `GeometricField::correctBoundaryConditions()` member function.

The boundary field itself is declared as a nested class template declaration, and the `GeometricField` stores it as a private attribute. The boundary field class template is named `GeometricBoundaryField`, its declaration is found within the `GeometricField` class template and it contains a list of boundary patch fields, one patch field per each mesh boundary. Although the `GeometricBoundaryField` is encapsulated in `GeometricField`, a non-constant access is provided, which results in the ability of client code of `GeometricField` to change the value of the boundary fields. At the first glance, this breaks encapsulation of the boundary fields by the geometric field, however, the benefit outweighs the design principles, because this approach results in a much greater usage flexibility. For example, a thermodynamical model may alter field values in a way that is dependent on another geometric field. The change in this case is driven from outside `GeometricField`, as will often be the case, as the geometric fields are global variables in OpenFOAM operated on by the solver implemented as a procedural sequence of calculations. Non-const access to the `GeometricBoundaryField_` attribute is shown in listing 71.

In contrast to the boundary field, the internal field is handled differently within the `GeometricField`. As the `GeometricField` is derived from `DimensionedField`, there is no need to return a different object. To access the internal field, a reference to `*this` is returned, as the `DimensionedField` implements the internal field and its arithmetic operations with dimension checking. Listing 72 clarifies how the geometric field provides access to the internal field. The `internalField()` member function returns the non-const reference to the `GeometricField` which,

Listing 71 Non-const access to the boundary fields in GeometricField.

```
private:
    //- Boundary Type field containing boundary field values
    GeometricBoundaryField GeometricBoundaryField_;

    // -- Some lines are missing --
public:
    //- Return reference to GeometricBoundaryField
    GeometricBoundaryField& GeometricBoundaryField();

    //- Return reference to GeometricBoundaryField for const field
    inline const GeometricBoundaryField& GeometricBoundaryField() const;
```

by inheritance, is a DimensionedField.

Listing 72 GeometricField providing access to internal dimensioned field.

```
// GeometricField.H
//- Return internal field
InternalField& internalField();

// GeometricField.C
template
<
    class Type,
    template<class> class PatchField,
    class GeoMesh
>
typename
Foam::GeometricField<Type, PatchField, GeoMesh>::InternalField&
Foam::GeometricField<Type, PatchField, GeoMesh>::internalField()
{
    this->setUpToDate();
    storeOldTimes();
    return *this;
}
```

At this point you should have an overview of the fields involved in the simulation in OpenFOAM and why the GeometricBoundaryField is encapsulated in the GeometricField, with non-constant access prepared for manipulation done by client code of the GeometricField class template. Analyzing the class inheritance and collaboration diagrams, although certainly helpful, is not as efficient in gaining understanding as using the classes themselves, which is addressed in the following sections.



10.2.2 Boundary Conditions

Boundary conditions, as their name implies, add functionality (conditional modification) to values stored in the `GeometricBoundaryField`. In Object Oriented Design (OOD), adding functionality usually implies extending existing classes, which is done in the case of boundary conditions as well. In fact, the `GeometricBoundaryField` described above does not encapsulate only field values stored at the domain boundary, each of the fields are extended with a function that modifies their values in a specific way - thus implementing a specific boundary condition.

The boundary conditions represent a hierarchical concept in the FVM - similar boundary conditions are grouped into boundary condition categories. For this reason, and to enable the user to select the boundary condition at runtime (RTS), they are modelled as a class hierarchy. The top parent abstract class `fvPatchField` defines the class interface to which each boundary condition must conform. Every boundary condition in OpenFOAM is either derived from `fvPatchField` or `pointPatchField`. The latter is mostly used for applications involving mesh motion or modification. Both have a constant private attribute that is called `internalField_` and is a reference to the internal field of the `GeometricField`, that was introduced in the previous section. The attribute provides access to values of the internal field, not only to the cells that are directly adjacent to the boundary mesh patch. In case of a `pointPatchField`, the `internalField_` attribute is declared as:

```
const DimensionedField<Type, pointMesh>& internalField_;
```

The declaration of the internal field for a `fvPatchField` is the same as for the `fvPatchField`, but the second template argument to `DimensionedField` is `volMesh`, rather than `pointMesh`:

```
const DimensionedField<Type, volMesh>& internalField_;
```

Since the `pointPatchField` conforms to the same class interface as the `fvPatchField`, and the volume fields are most often encountered, the `fvPatchField` is covered in this section.

Before we go into detail about which member function of `fvPatchField`

is of relevance, when it comes to the implementation of a new boundary condition, we try to conclude the overview of the connection between the `GeometricField` and the actual access to the boundary conditions. A graphical representation of this relation is given in figure 10.2. The geometrical field composes the geometrical boundary field, which inherits from (`FieldField`) and therefore is a collection of (boundary) fields. The composition of the geometrical boundary field is required because the modification of the internal field value requires an update of the boundary field values by the boundary conditions. Also, the boundary fields cannot be separated into objects distinct from the internal field. Internal and boundary fields are not only topologically attached to each other, via the mesh, the FVM requires the boundary field values when the equation is discretized to compute internal field values. The refinement of the mesh causes splitting of cell faces, so the lengths of internal and boundary fields are indirectly connected in this case as well. Having separate internal and boundary fields therefore would make no sense at all. It would introduce global variables that need to be explicitly synchronized, which would severely complicate the semantics of all field operations on the application level. The geometric field loops over the collection of boundary fields and updates each boundary field by delegating the update to the corresponding boundary condition. Figure 10.2 shows `PatchField` template parameter which, when instantiated (`fvPatchField` for a volume mesh), is a boundary condition.

The solver usually calls the `correctBoundaryConditions()` member function of the `GeometricField` when the internal field is modified and the boundary conditions are to be updated. The implementation of the `GeometricField::updateBoundaryConditions()` member function is shown in listing 73.

WARNING

To understand how boundary conditions are updated by the `GeometricField` the last line in listing 73 must be understood.

The last line in listing 73 calls the `evaluate()` member function of the `GeometricBoundaryField` that in turn performs a variety of tasks. This member function calls the `initEvaluate()` member function of



Listing 73 The `GeometricField::updateBoundaryConditions()` member function.

```
// Correct the boundary conditions
template
<
  class Type,
  template<class> class PatchField,
  class GeoMesh
>
void Foam::GeometricField<Type, PatchField, GeoMesh>::
correctBoundaryConditions()
{
    this->setUpToDate();
    storeOldTimes();
    GeometricBoundaryField_.evaluate();
}
```

`fvPatchField`, if the boundary condition has not been initialized. Otherwise the `evaluate()` member function is called. Due to the zero halo layer parallelism implemented by OpenFOAM, the parallel communication is handled by `GeometricBoundaryField::evaluate()`, as the process boundaries are also implemented as boundary conditions.

The `GeometricBoundaryField::updateCoeffs()` member function is the other major member function that triggers functionalities of the particular `fvPatchField` from the client code. Compared to `evaluate()`, the implementation of `updateCoeffs()` is shorter since no parallel communication is implemented. The implementation of the `GeometricBoundaryField::updateCoeffs` is shown in listing 74.

The `forAll` loop in listing 74 loops over all patches of the `GeometricBoundaryField` which is referred to as `*this`. The `updateCoeffs()` member function of `fvPatchField` is called directly for each element of the domain boundary, using the operator `[]`.

Despite some operators, the `updateCoeffs()` and `evaluate()` member functions represent the entire public interface to the `fvPatchField` that is accessed automatically from each solver. The major difference between both member functions is that `evaluate()` can be called an arbitrary number of times in a single time step, but is only executed once. On the other hand, `updateCoeffs()` does not check if the boundary condition

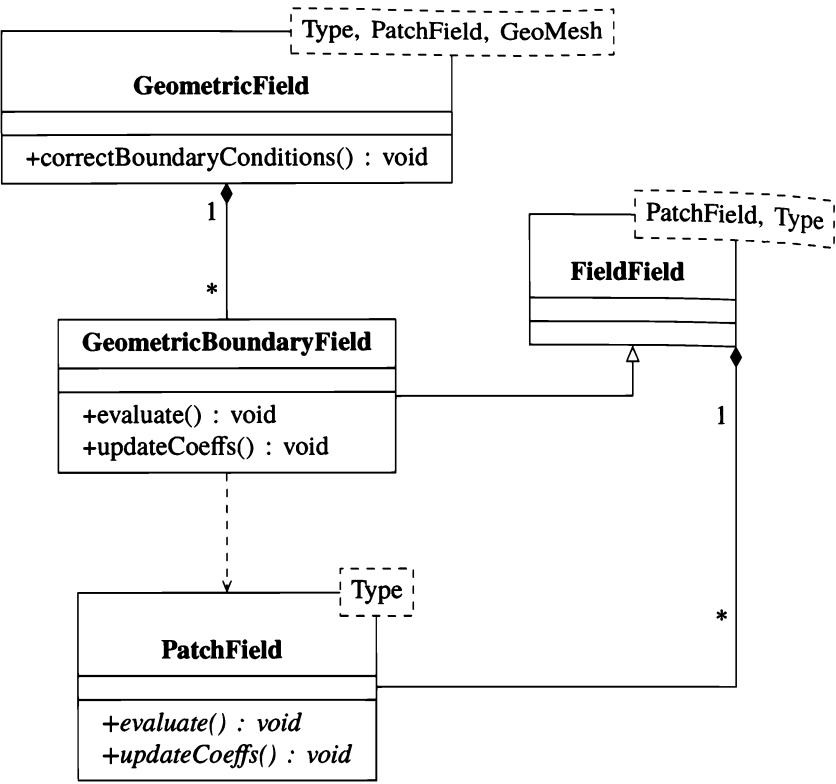


Figure 10.2: Class collaboration diagram for the boundary conditions.

is updated or not, it will perform the calculation as many times as it is called. Both are part of the general class interface provided by the base class `fvPatchField`, that can be used to program a customized boundary conditions, derived either directly or indirectly from `fvPatchField`.

Only a small number boundary conditions in the official release are derived directly from `fvPatchField`, such as the basic `fixedValueFvPatchField` and `zeroGradientFvPatchField`.

Most of the derived boundary conditions inherit directly from basic boundary conditions. A popular base class is `mixedFvPatchField` which provides the functionality of blending between a user defined fixed value and a fixed gradient boundary condition. The class collaboration diagram



Listing 74 The `GeometricBoundaryField::updateCoeffs` member function.

```

template
<
  class Type,
  template<class> class PatchField,
  class GeoMesh
>
void
Foam::GeometricField
<
  Type,
  PatchField,
  GeoMesh
>::GeometricBoundaryField::updateCoeffs()
{
    if (debug)
    {
        Info<< "GeometricField<Type, PatchField, GeoMesh>::"
              "GeometricBoundaryField::"
              "updateCoeffs()" << endl;
    }

    forAll(*this, patchi)
    {
        this->operator[](patchi).updateCoeffs();
    }
}

```

of `mixedFvPatchField` is shown in figure 10.3. It contains three new private attributes shown also in listing 75, and it does not rely on the implementation of the fixed gradient and fixed value boundary conditions. Instead, the prescribed fixed gradient and fixed value boundary fields are stored as private attributes of type `Field`.

As these attributes are private, there are public member functions that provide const and non-const access to them. This enables the derived classes to use the attributes indirectly in order to implement different ways of blending between the fixed value and zero gradient boundary condition. A commonly used boundary condition that is derived directly from `mixedFvPatchField` is the `inletOutletFvPatchField`. It switches between fixed value and zero gradient boundary condition, depending on the direction of the flux. If the flux is pointing out of the domain, it acts just as the zero gradient boundary condi-

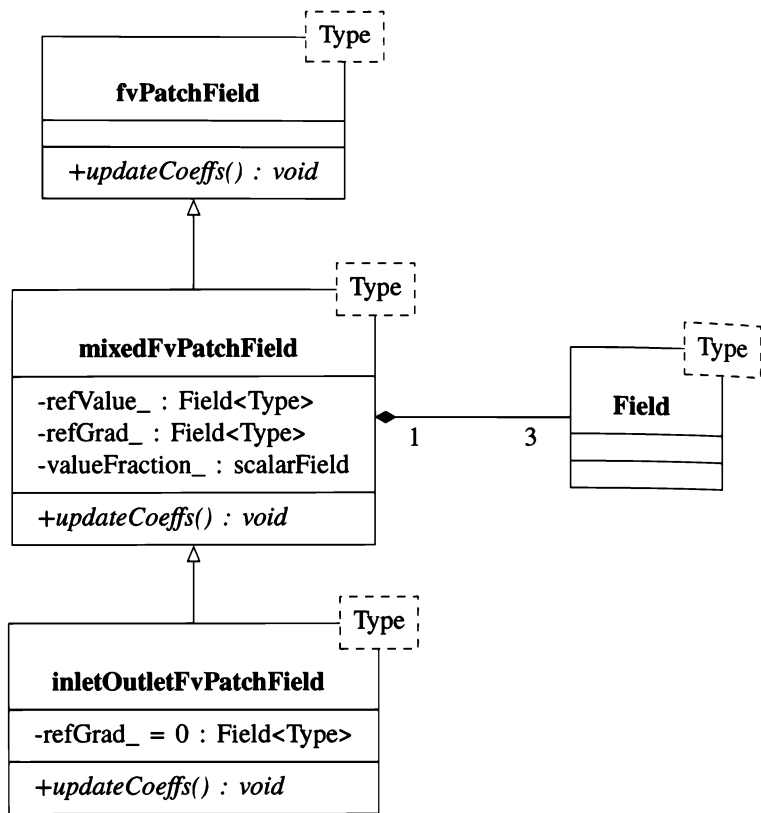


Figure 10.3: Class collaboration diagram for `mixedFvPatchField` and `inletOutletFvPatchField` boundary conditions.

WARNING

The numerical background of the fixed value and the zero gradient boundary condition is covered in section 1.3.

tion (`zeroGradientFvPatchField`), otherwise it acts as a fixed value boundary condition (`fixedValueFvPatchField`). This is determined on a face-by-face basis, and based on the private attributes of the `mixedFvPatchField` boundary condition. The gradient of the field is not prescribed by the user as in the `mixedFvPatchField` - it is set to the value of zero by the `inletOutletFvPatchField` constructor.



Listing 75 Private attributes of the `mixedFvPatchField` boundary condition.

```

//- Value field
Field<Type> refValue_;

//- Normal gradient field
Field<Type> refGrad_;

//- Fraction (0-1) of value used for boundary condition
scalarField valueFraction_;
  
```

The fraction value used later by `mixedFvPatchField<Type>::updateCoeffs()` is computed in listing 76 by assigning the value 1 for faces that have a positive (outflow) volumetric flux, and value 0 otherwise. The implementation of the `updateCoeffs()` member function of `inletOutletFvPatchField` is shown in listing 76. Only the values for the `valueFraction()` are set by this member function, the calculation of the boundary field is delegated to the parent `mixedValueFvPatchField`. The function `pos` is shown in listing 77. It returns 1 if the value of the scalar `s` is greater or equal than zero and 0 otherwise. The actual assignment of `zeroGradientFvPatchField` and `fixedValueFvPatchField` is done by the call to `mixedFvPatchField::updateCoeffs()` and hence must not be re-implemented.

Reading Boundary Condition Data

During the process of programming a custom boundary condition that may have new parameters, corresponding new parameter names and values need to be read from a field file in the 0 directory. Therefore, the programmer must add the necessary parameters and their values in that file. The as the boundary conditions are read from files in the form of dictionaries, the dictionary is read and passed to the boundary condition constructor.

Some boundary conditions may use the dictionary class member function that looks up data while providing default values. In that case, switching the type of the boundary condition and not providing the appropriate parameters will not result in a runtime error. Operations on the dictionary class are covered in chapter 5 and should be well understood before

Listing 76 The `updateCoeffs()` member function of the inlet/outlet boundary condition.

```
template<class Type>
void Foam::inletOutletFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const Field<scalar>& phip =
        this->patch().template lookupPatchField
        <
            surfaceScalarField,
            scalar
        >
        (
            phiName_
        );

    this->valueFraction() = 1.0 - pos(phpip);

    mixedFvPatchField<Type>::updateCoeffs();
}
```

Listing 77 The `pos` function.

```
inline Scalar pos(const Scalar s)
{
    return (s >= 0)? 1: 0;
}
```

proceeding to the next section where programming of a new boundary condition is explained.

10.3 Implementing a new Boundary Condition

The previous section provided an overview of the implementation of boundary conditions in OpenFOAM. In this section the implementation of two new boundary conditions is described.

OpenFOAM provides a large number of different boundary conditions to choose from, and community developments have been done in this part



of the code as well, with the most prominent one being the `groovyBC` boundary condition in the `swak4Foam` contribution¹. Before writing a new boundary condition that fits your requirements, it is wise to take a look if this functionality is already available in the code base or can be modelled by the `groovyBC` boundary condition.

There is quite a large amount of information on how to write your own boundary condition in OpenFOAM already available on the internet. Examples in this chapter have been prepared independent of the already available material. The first example shows how to extend the functionality of any boundary condition in OpenFOAM without modifying it, with a concrete purpose of reducing recirculation at the boundary. The second example shows how develop a new `pointPatchField` that applies a predefined motion to a patch. This predefined motion is calculated from tabulated data, that must be provided by the user. Both examples emphasise the correct use of the class interface fixed by the root abstract base classes for the boundary conditions in OpenFOAM, i.e. the `fvPatchField` and `pointPatchField` classes.

10.3.1 Recirculation Control Boundary Condition

In this example a method is presented how an existing boundary condition can be extended by an additional computation (functionality) during runtime. Imagine a boundary condition that updates the boundary field values in a certain way as a simulation runs. At some point, based on the simulation results, the conditions of the simulation (e.g. a pressure at another boundary) signal that an additional boundary operation is necessary. When this happens, the new extended boundary condition *enables* the additional calculation at runtime, and modifies the boundary field values accordingly.

A good technical example would be a heated closed container filled with an ideal gas. When heat flux enters the container, the pressure in the container rises. An extended boundary condition would measure the pressure at the lid of the container, and open the lid when the pressure reaches a certain value. Numerically speaking, this boundary condition would *change its type* during the simulation, from an impermeable wall,

¹<http://openfoamwiki.net/index.php/Contrib/swak4Foam>

to an outlet boundary condition, based on the pressure at the lid.

In the example presented in this chapter, recirculation is measured at a boundary condition. When recirculation occurs, the additional calculation modifies the nature of the boundary condition into an *inflow* condition. As a result, the recirculation is *pushed out* by the inflow.

WARNING

The recirculation boundary condition example may not work in a parallel execution, depending on the fact if the modified boundary is a part of the processor domain. The goal of this example is to show how the mesh, the object registry and the fields collaborate with each other, and not how to parallelize non-standard boundary condition updates.

Hypothetically, such an extension could be achieved using inheritance only. However, it would require an extension of each boundary condition in OpenFOAM to account for the RTS-enabled additional calculation that needs to be performed. Extending each boundary condition by using multiple inheritance would result in modifications of the existing boundary conditions, only to take into account *possible extension* (i.e. recirculation control) which may or may not be used at runtime, depending on the user's choice. Obviously, this is by no means a versatile method and it makes the extension difficult to achieve during runtime without modifying existing code. When a new functionality needs to be added during runtime to an already existing hierarchy of classes without modifying the models of that hierarchy, the object oriented design pattern *Decorator Pattern* can be used. Details on the Decorator Pattern and other OOD patterns are given in the book by Gamma, Helm, Johnson, and Vlissides 1995. Figure 10.4 illustrates the working principle for a `zeroGradient` boundary condition, that is decorated with an arbitrary functionality.

Adding functionality to a BC using the Decorator Pattern

A boundary condition decorator *is* a boundary condition itself, since it modifies the boundary field as well. Therefore, the decorator inherits from the `fvPatchField` abstract base class of all boundary conditions in OpenFOAM. In addition to inheriting from the `fvPatchField`, the decorator composes object of it's own base class (`fvPatchField`).



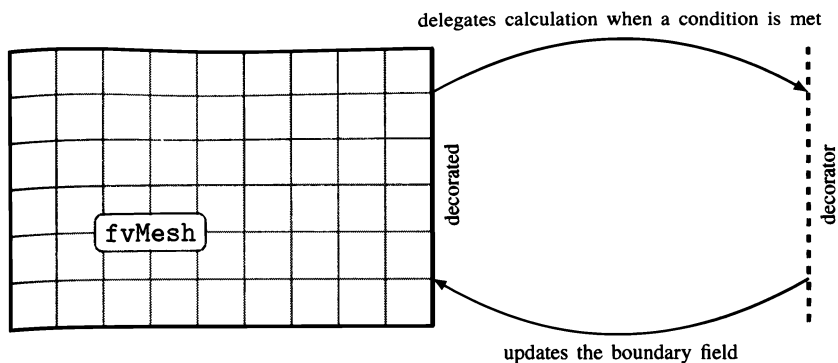


Figure 10.4: Working principle of the decorated boundary condition. The boundary condition operates in a standard way until such conditions are fulfilled that the extended computation is required and delegated to the decorator. At that point, the boundary condition acts as if it switched to the decorator type.

To clarify things, the UML class collaboration diagram of the Decorator Pattern applied on the boundary conditions hierarchy is shown in figure 10.5. As shown on in that diagram, the decorator is a part of the class hierarchy as other boundary conditions. Hence, for boundary conditions in OpenFOAM, imposing such an *is-a* relationship with the abstract base class `fvPatchField` makes the boundary condition decorator act as a boundary condition to the rest of the OpenFOAM client code. This is exactly the same principle as plain inheritance, only with the extension of storing an instance of the object inherited from. The decorator must implement all the pure virtual methods prescribed by the abstract class `fvPatchField`. As it composites an ordinary boundary condition as well, the decorator will delegate the function calls to the decorated boundary condition, based on the conditions prescribed by the programmer. The extensions provided by the decorator can be designed as the programmer desires. By combining both inheritance and composition as shown in figure 10.5, it is possible for a decorated boundary condition to switch its own type during runtime, as it delegates the computation to the decorator.

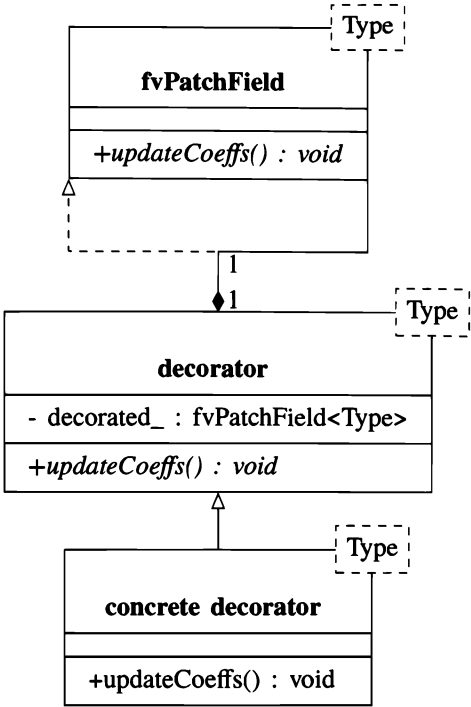


Figure 10.5: Decorator for the boundary conditions in OpenFOAM

Adding recirculation control to a boundary condition

As an example of adding runtime functionality to any boundary condition in OpenFOAM we have chosen to flow recirculation as the control parameter and inflow velocity as the imposed action of the boundary condition. The flow recirculation is recognized by an alternating sign the volumetric flux at the boundary. Alternating sign in the volumetric flux on a domain boundary signals an that an eddy (vortex) crosses the boundary. Therefore, the fluid flows into the domain over a part of the boundary, and flows out of the over the other boundary part.

For the example simulation case, the boundary condition checks if the flow of the extended boundary has recirculation and tries to control the decorated boundary condition in order to reduce it. This type of boundary condition decorator is most likely to be set for some of the boundary conditions where outflow is expected. This control in this example



is done in a fairly straightforward manner: by modifying the decorated boundary condition as such that its field is overwritten with increasing inflow values.

Of course, this example is specific to inflow/outflow situations, but the goal of this example is not to deal with flow control. The recirculation control boundary condition decorator is different than the standard boundary conditions in OpenFOAM. It modifies field values calculated by another boundary condition directly, regardless of the type of boundary condition that is decorated.

WARNING

Boundary condition decorator is not a working solution to an actual CFD problem, it merely represents an interesting example for programming new boundary conditions in OpenFOAM.

A finished recirculation control boundary condition is already available in the example code repository. To ease the understanding of the presented example, the description should be followed with the code of the recirculation control boundary condition from the example code repository opened in a text editor. In the following discussion the same names are used for the files and classes as those present in the example code repository.

TIP

The boundary conditions have to be compiled as shared libraries. They should never be implemented directly into application code, as this limits their usability significantly as well as sharing with other OpenFOAM programmers.

The libraries that the boundary conditions are compiled into are dynamically linked with solver applications during runtime. At the start of the tutorial, the library directory needs to be created:

```
?> mkdir -p primerBoundaryConditions/recirculationControl  
?> cd !$
```

and the class files of an existing boundary condition need to be copied,

which we will use as skeleton files for the recirculation control boundary condition:

```
?> cp \
$FOAM_SRC/finiteVolume/fields/fvPatchFields/basic/zeroGradient/* .
```

The dependency files generated during compilation are best to be removed at this point:

```
?> rm *.dep
```

All instances of the string "zeroGradient" are to be renamed into "recirculationControl" in file names as well as in class names. Once the names are modified, exit the `recirculationControl` directory, and create the compilation configuration folder `Make`:

```
?> cd ..
?> wmakeFilesAndOptions
```

and modify the `Make/files` to take into account that a library is to be compiled, and not an application, so the line

```
EXE = $(FOAM_APPBIN)/primerBoundaryConditions
```

needs to be replaced with the line

```
LIB = $(FOAM_USER_LIBBIN)/libprimerBoundaryConditions
```

Note that the script `wmakeFilesAndOptions` will insert all `*.C` files into the `Make/files` file, which means that the line

```
recirculationControl/recirculationControlFvPatchField.C
```

must be removed before compilation, since it contains class template definition for the recirculation control boundary condition. This will cause class re-definition errors during compilation, since the actual boundary condition classes are instantiated from the recirculation control class template for the tensorial properties using the macro

```
makePatchFields(recirculationControl);
```



Once the file `recirculationControlFvPatchField.C` is removed from `Make/files`, the library is ready for the first compilation test. The compilation can be started by issuing

```
?> wmake libso
```

from within the `primerBoundaryConditions` directory.

Listing 78 The `Make/files` file of the `primerBoundaryConditions` library.

```
recirculationControl/recirculationControlFvPatchFields.C

LIB = $(FOAM_USER_LIBBIN)/libprimerBoundaryConditions
```

Listing 79 The `Make/options` file of the `primerBoundaryConditions` library.

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude

LIB_LIBS = \
    -lfiniteVolume
```

Listing 78 contains final version of the `Make/files` and listing 79 contains the final version of the `Make/options` file. With the successful compilation, a skeleton implementation of a self-contained shared boundary condition library is created. Before further development of the decorator functionality is started, the boundary condition is to be tested with an actual simulation run. At this point the boundary condition has the same functionality as the `zeroGradientFvPatchField` which it is based upon, but with a different name. It's functionality can be tested by running any simulation case, provided you define the required linked libraries in the `system/controlDict` file:

```
libs ("libprimerBoundaryConditions.so")
```

This loads the boundary condition shared library and links it to the

OpenFOAM executable.

TIP

Performing such integration tests is a strongly recommended step in the process of writing a custom library. Using a version control system (Chapter 6) improves the workflow as well. As an example, try applying the recirculation control boundary condition to the `fixedWalls` patch of the velocity field in the cavity tutorial case using the `icoFoam` solver.

At this point, the integration of the library with OpenFOAM is tested and considered working. The implementation of the design shown in figure 10.5 may begin. The strict object oriented way of implementing the Decorator Pattern would be to start with an abstract Decorator class for the `fvPatchField`. In that case, the `recirculationControl` must be derived from it as a concrete decorator model. Instead, a concrete decorator is used in this example as a starting point and the abstract implementation is left to the reader as an exercise at the end of the section. Implementing the exercise allows the addition of different functionalities to any boundary condition in OpenFOAM without modifying their implementation. This is an additional benefit of doing the exercise besides improving the understanding of boundary conditions in OpenFOAM.

The first modification to the recirculation control boundary condition is applied to the class template declaration file `recirculationControlFvPatchField.H`. It inherits from `fvPatchField`, which is done by `zeroGradientFvPatchField` already. The important parts of `recirculationControlFvPatchField`'s class definition are shown in code listings.

In listing 80, the decorator pattern can be seen: the inheritance from `fvPatchField` is present but also a private attribute of `fvPatchField` is composited by the recirculation control boundary condition class. Instantiating the local copy in `baseTypeTmp_` is done using the smart pointer object `tmp` of OpenFOAM, as it provides additional functionalities such as garbage collection. This smart pointer relies on the RAII C++ idiom and greatly simplifies the handling of pointers. The remaining class attributes that are declared as constant will be read from the boundary



Listing 80 The declaration of the recirculation control boundary condition.

```

template<class Type>
class recirculationControlFvPatchField
:
    public fvPatchField<Type>
{
    protected:

        // Base boundary condition.
        tmp<fvPatchField<Type> > baseTypeTmp_;

        const word applyControl_;
        const word baseTypeName_;
        const word fluxFieldName_;
        const word controlledPatchName_;
        const Type maxValue_;

        scalar recirculationRate_;

```

condition's dictionary. In order to instantiate the boundary condition that should be decorated, the particular name must be passed via the boundary condition's dictionary and is stored in the `baseTypeName_` attribute. To provide an option to either apply the control of the recirculation or not, the Switch `applyControl_` is implemented. If it is false, nothing will be imposed upon the boundary, and so switching it off and simply using the decorated boundary condition is easy. In addition it will report the amount of recirculation on the extended boundary condition. The type of the extended boundary condition is determined by the `baseTypeName_` variable, which is then used to create a concrete extended boundary condition based on a class name. This boundary condition is stored in `baseTypeTmp_`.

In order to compute the recirculation, the boundary condition needs to know the name of the volumetric flux field which is what the `fluxFieldName_` attribute defines. The name of the patch field which is controlled is defined by the `controlledPatchName_`. The maximal value it may assume is defined by the member `maxValue_`, and the `recirculationRate_` stores the percentage of the negative volumetric flux on the extended boundary condition.

Having added the new class attributes, the constructors of `recirculationControlFvPatchField` will have to initialize them. The common is to expand and modify the constructor initialization lists to take into account the new private class attributes. To keep the description concise, only the dictionary constructor is shown in the following text. The full implementation is available in the source code repository.

Listing 81 Constructor of the recirculation control boundary condition.

```
template<class Type>
Foam::recirculationControlFvPatchField<Type>::
recirculationControlFvPatchField
(
    const recirculationControlFvPatchField<Type>& ptf,
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fvPatchField<Type>(ptf, p, iF, mapper),
    baseTypeTmp_(),
    applyControl_(ptf.applyControl_),
    baseTypeName_(ptf.baseTypeName_),
    fluxFieldName_(ptf.fluxFieldName_),
    controlledPatchName_(ptf.controlledPatchName_),
    maxValue_(ptf.maxValue_),
    recirculationRate_(ptf.recirculationRate_)
{
    // Instantiate the baseType based on the dictionary entries.
    baseTypeTmp_ = fvPatchField<Type>::New
    (
        ptf.baseTypeTmp_,
        p,
        iF,
        mapper
    );
}
```

The constructor of the recirculation control boundary condition is shown in listing 81. In the initialization list, `baseTypeTmp_` is not assigned a value and hence takes an arbitrary value. In the constructor itself, a `fvPatchField` is created and assigned to `baseTypeTmp_`, using the `New` selector. This way of constructing objects is called `Factory Method` (`New Selector`) and is defined in the abstract `fvPatchField` class. It selects the base class for `recirculationControlFvPatchField` during runtime, based on the name provided in the dictionary `dict` is used by the constructor to initialize the decorated boundary condition. After



this constructor has been executed, the protected attributes needed by the control function will be initialized, as will the decorated boundary condition.

Listing 82 Recirculation control boundary condition delegating a member function call.

```
template<class Type>
Foam::tmp<Foam::Field<Type> > >
Foam::recirculationControlFvPatchField<Type>::valueInternalCoeffs
(
    const Foam::tmp<Foam::Field<scalar> > & f
) const
{
    return baseTypeTmp_->valueInternalCoeffs(f);
}
```

The remaining constructors of `recirculationControlFvPatchField` must be modified similarly, accounting for the new class attributes. Once the constructors are modified, the member functions listed in figure 10.5 need to be modified as well. They need to delegate the work to the decorated boundary condition, that is stored in `baseTypeTmp_`. The implementation is identical for all four member functions, so only one of them is presented in listing 82.

This kind of delegation needs to be present in all parts of the decorator implementation where the control of the flow *is not performed*. In that case the boundary condition should act as the decorated boundary condition. This is illustrated as the concrete boundary condition model in figure 10.5.

As the final step of implementing the recirculation control boundary condition, the member function responsible for the boundary condition operation (`updateCoeffs`) needs to be implemented. The implementation is divided into two major parts. The first part checks if the boundary condition has already been updated, which is common practice in OpenFOAM boundary conditions. If this is not so, the flux field is looked up using the user-defined name of the flux field `fluxFieldName_`.

The positive and negative volumetric fluxes are computed by the extended boundary condition as shown in listing 83. Once the positive and

Listing 83 Computing positive and negative fluxes by the recirculation boundary condition.

```
if (this->updated())
{
    return;
}

typedef GeometricField <Type, fvPatchField, volMesh> VolumetricField;

// Get the flux field
const Field<scalar>& phip =
this->patch().template lookupPatchField
<
    surfaceScalarField,
    scalar
>(fluxFieldName_);

// Compute the total and the negative volumetric flux.
scalar totalFlux = 0;
scalar negativeFlux = 0;

forAll (phip, I)
{
    totalFlux += mag(phip[I]);

    if (phip[I] < 0)
    {
        negativeFlux += mag(phip[I]);
    }
}
```

negative volumetric fluxes are computed, the recirculation rate (ratio of the negative flux and the total flux) is calculated with the code shown in listing 84.

In conditions where no recirculation occurs, the recirculation control and the decorated boundary condition behave identically. The modification of the field is thus delegated to the encapsulated concrete boundary condition model. Note that because the boundary condition decorator is itself a boundary condition, each time an update happens, the boundary condition state must be set to up-to-date. As usual the member function `updateCoeffs` of the `fvPatchField` abstract class must be invoked for this purpose. The part of the `updateCoeffs` member function that is responsible for the control of the recirculation is shown in listing 85.



Listing 84 Computing recirculation rate on the boundary.

```

// Compute recirculation rate.
scalar newRecirculationRate = min
(
    1,
    negativeFlux / (totalFlux + SMALL)
);

Info << "Total flux " << totalFlux << endl;
Info << "Recirculation flux " << negativeFlux << endl;
Info << "Recirculation ratio " << newRecirculationRate << endl;

// If there is no recirculation.
if (negativeFlux < SMALL)
{
    // Update the decorated boundary condition.
    baseTypeTmp_->updateCoeffs();
    // Mark the BC updated.
    fvPatchField<Type>::updateCoeffs();
    return;
}

```

This boundary condition will need a non-constant access to the field of another boundary condition, since it will modify boundary field values. For this reason, it *breaks the encapsulation* of the `objectRegistry` class, by casting away constness of the `VolumetricField` provided by the registry forcefully. The non constant access to the other boundary field is shown in listing 86.

WARNING

Casting away constness this way should be avoided wherever possible. It invalidates the point of encapsulation - the object state that can be modified only by the class member functions. This example performs this kind of a cast to emphasize the class collaboration between the geometrical field and the object registry.

If the interface of the `objectRegistry` provides only constant access to registered object, casting away constness and modifying the object state is deceiving the programmer. He/she will not expect the change of the `VolumetricField` object state to be possible via the interface of the `objectRegistry`. Algorithm 1 clarifies the recirculation control algorithm in pseudocode.

This boundary condition is meant primarily as an example of applying the Decorator Pattern to the boundary condition class hierarchy in OpenFOAM. However, the design applied here may very well be used in situations where inlet/outlet boundary conditions are present. In some situations a reduction of recirculation can be achieved by increasing e.g. the inlet pressure or velocity. Still, note that this is only an example that illustrates two major things: First it shows how the design of the `VolumetricField` may be used to couple functionality between different boundary conditions. Secondly, it illustrates how to program a new boundary condition in OpenFOAM, that is derived from `fvPatchField`.

Testing the recirculation control boundary condition

Once the `updateCoeffs` method has been implemented, the boundary condition is ready to be used. As a test case, a simple backward channel with a backward facing step is used. The recirculation control is applied on the outlet of the channel, with the backward facing step wall as the controlled boundary condition.

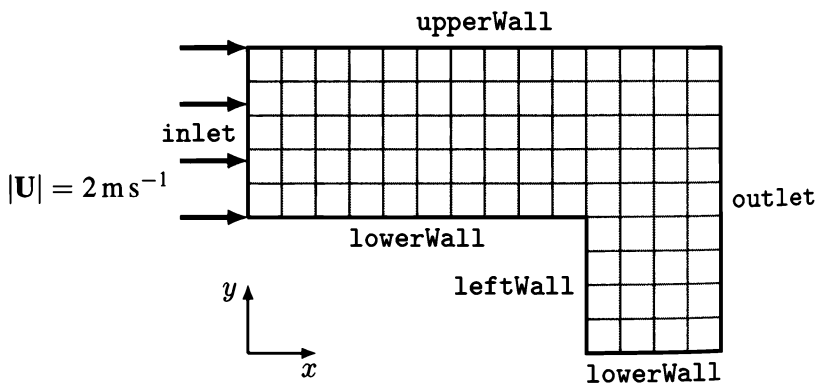
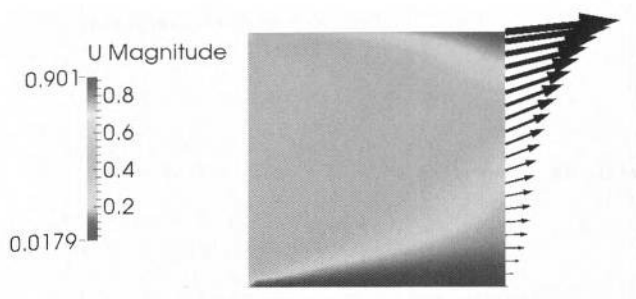


Figure 10.6: Geometrical setup and initial conditions for the recirculation control test case

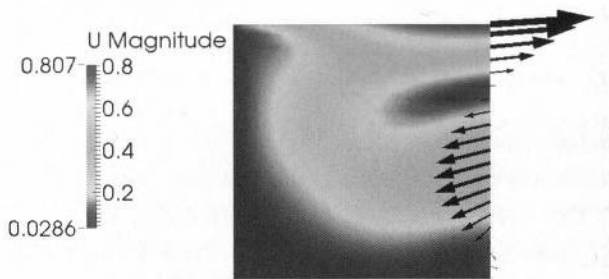
Figure 10.6 depicts the initial configuration of the flow, where the velocity of the backward step is set to zero as it is an impermeable wall. The recirculation control boundary condition will override the zero velocity of the backward facing step the moment recirculation appears at the outlet. Further on, it will switch the boundary condition applied to the `leftWall` boundary into an inflow, in order to drive the recirculation out. The solver



`icoFoam` is used for this simulation of real-time laminar flow control in a channel.



(a) Recirculation controlled.



(b) Recirculation not controlled.

Figure 10.7: Final velocity field for the recirculation control test case.

The final velocity field with the recirculation control boundary condition disabled is shown in figure 10.7b. Figure 10.7a shows the velocity field without recirculation occurring at the outlet. In this case the recirculation control boundary condition was imposing control on the `leftWall` boundary. Both the `recirculationChannel` and `recirculationControlChannel` test cases are located in the `chapter10` sub-directory of the example case repository.

EXERCISE

An abstract Decorator was not implemented in this example. Modify the `recirculationControlFvPatchField` such that an abstract Decorator is added to the class hierarchy. That decorator generalizes the decoration of the boundary condition, resulting in the ability to add any functionality to any boundary condition in OpenFOAM at runtime.



Listing 85 Recirculation control execution.

```

if (
    (applyControl_ == "yes") &&
    (newRecirculationRate > recirculationRate_)
)
{
    Info << "Executing control..." << endl;

    // Get the name of the internal field.
    const word volFieldName = this->dimensionedInternalField().name();

    // Get access to the registry.
    const objectRegistry& db = this->db();

    // Find the GeometricField in the registry using
    // the internal field name.
    const VolumetricField& vfConst =
        db.lookupObject<VolumetricField>(volFieldName);

    // Cast away constness to be able to control
    // other boundary patch fields.
    VolumetricField& vf = const_cast<VolumetricField&>(vfConst);

    // Get the non-const reference to the boundary
    // field of the GeometricField.
    typename VolumetricField::GeometricBoundaryField& bf =
        vf.boundaryField();

    // Find the controlled boundary patch field using
    // the name defined by the user.
    forAll (bf, patchI)
    {
        // Control the boundary patch field using the recirculation rate.
        const fvPatch& p = bf[patchI].patch();

        if (p.name() == controlledPatchName_)
        {
            if (! bf[patchI].updated())
            {
                // Invoke a standard update first to avoid the field
                // being later overwritten.
                bf[patchI].updateCoeffs();
            }
            // Compute new boundary field values.
            Field<Type> newValues (bf[patchI]);

            scalar maxNewValue = mag(max(newValues));

            if (maxNewValue < SMALL)
            {
                bf[patchI] == 0.1 * maxValue_;
            } else if (maxNewValue < mag(maxValue_))
            {
                // Impose control on the controlled inlet patch field.
                bf[patchI] == newValues * 1.01;
            }
        }
    }
}

```



Listing 86 Getting non-const access to another boundary field, from the recirculation boundary condition.

```
// Cast away constness to be able to control other boundary patch fields.
VolumetricField& vf = const_cast<VolumetricField&>(vfConst);

// Get the non-const reference to the boundary field of the GeometricField.
typename VolumetricField::GeometricBoundaryField& bf = vf.boundaryField();
```

Algorithm 1 Recirculation control algorithm.

```
if control is applied and recirculation is increasing then
  get the name of the controlled field
  get access to object registry
  find the VolumetricField in the object registry
  cast away constness of the VolumetricField
  for boundary conditions do
    if controlled boundary condition found then
      compute the new values
      if new values are zero then
        set new values to 10% of the maximum
      else if new values are positive and smaller than prescribed maxi-
    mum then
      increase the old values by 1 %
    end if
  end if
end for
end if
```



10.3.2 Mesh Motion Boundary Condition

This sub-section illustrates the construction of a new boundary condition used for the motion of the mesh. The motion of the mesh relies on displacements or velocities defined for the points of the mesh, so this boundary condition relates to the mesh boundary described as a set of boundary points (a `pointPatchField`).

Unlike the boundary conditions that are based on the `fvPatchField`, `pointPatchField`-type boundary conditions do not store the boundary values in a boundary field, they are used to modify the values of the internal field. Altering the internal field values and any other operations are handled by either the solver or other classes using these boundary conditions. Vector quantities used by the mesh motion boundary conditions will either define a velocity or a displacement of the particular mesh point, depending on the choice of the mesh motion solver.

A mesh motion solver will use the `dynamicFvMesh` library and in the case of the example presented in this sub-section, it will solve a Laplacian equation for the point displacement with the displacement at the boundary defined by the new boundary condition described in this sub-section. Since the Laplacian equation is used to model diffusive radial transport, the displacement prescribed at the mesh boundary will be smoothly diffused to the surrounding mesh, which ensures higher quality of the deformed cells. For this type of application, the field that stores the deformation of the points is called `pointDisplacement`.

The boundary condition presented in this section reads the position and orientation of the patch's centre of gravity from an input file and applies the displacement to the mesh boundary, with respect to its previous position. It must be used in conjunction with dynamic meshes, otherwise the `pointDisplacement` field will not be read. The boundary condition functionality consists of two major components, each of which is already existent in various places of the OpenFOAM release:

1. Computing the position of the patch's centre of gravity (COG), which is calculated based on the prescribed motion that is contained in a dictionary. This prescribed motion must be present in tabulated form and gets interpolated linearly between each of the data points.

All of this is already implemented in the `tabulated6DoFMotion`, which is a `dynamicFvMesh` class that moves the entire mesh based on the prescribed motion.

2. Assigning vectorial values to a `pointPatchField`. One example for this type of boundary condition is the `oscillatingDisplacementPointPatchVectorField`.

As for boundary conditions derived from `fvPatchField`, only values on the domain boundary are changed. No additional changes to the simulation or field are performed by any boundary condition and the functionalities are encapsulated in a logical manner. For `fvPatchField` boundary conditions, the field variables are calculated by the flow solver, using the boundary values. The same principle applies to boundary conditions derived from `pointPatchField`. Only the velocity or displacement is prescribed by the boundary condition, whereas the actual mesh changes are performed by a dedicated mesh motion solver that is a part of the `dynamicFvMesh` library. In the following, both components are described in a brief manner and the parts that are relevant for the new boundary condition are highlighted.

Reading the Motion Data

A brief search in the existing code base of OpenFOAM reveals that there is a mesh motion solver that reads the motion data from a tabulated file, similar to what is planned for this boundary condition. However, in that case, the same displacement is applied to all mesh points, resulting in a mesh motion which moves the mesh as a rigid body: no mesh deformation occurs and the relative position of the mesh points does not change. For the purpose of this tutorial, the calculation of motion can be used and later be applied to the patch points making the boundary of the mesh move as a rigid body. This kind of mesh motion may be beneficial when the relative motion of the body is small with respect to the mesh (flow domain). In this case, if the motion is propagated into the flow domain in a diffusion-like manner, the motion of the mesh far away from the boundary with prescribed displacement will be near zero. How fast the displacement vanishes away from the body is determined by the magnitude of the displacement diffusion coefficient.

The code implementing the rigid body mesh motion based on tabular



data is contained within the `tabulated6DoFMotionFvMesh` class which is derived from `solidBodyMotionFunction` and can be found here:

```
?> $FOAM_SRC/dynamicFvMesh/solidBodyMotionFvMesh/\
    solidBodyMotionFunctions/tabulated6DoFMotion/
```

There is an example case in the official release that employs the `tabulated6DoFMotion` to prescribe the motion of a closed tank. This tutorial can be found here:

```
?> $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/sloshingTank3D6DoF
```

The dictionary containing the motion data (points in time) is set up as a list (using the `List` data structure) which is consisted of translation and rotation vectors in time t . An example for this data can be found in `constant/6DoF.dat` located in the tutorial mentioned above. The snippet below illustrates the principle of the way that dictionary is set up.

```
(
(t1 ((Translation_Vector_1) (Rot_Vector_1))
(t2 ((Translation_Vector_2) (Rot_Vector_2))
...
)
```

A spline based interpolation is performed to obtain position and orientation data between the data points of the dictionary. The translation and rotation vectors are defined with respect to the original coordinate system and as shown in figure 10.8.

As chapter 13 deals solely with dynamic meshes, only a brief summary of the working principles of dynamic meshes of type `solidBodyMotionFvMesh` is provided for better clarity:

- dynamic meshes that are derived from `solidBodyMotionFvMesh` deal only with the motion of solid bodies,
- no topological changes can be performed, nor is a solid body patch supposed to change it's shape,
- the motion itself is not defined by the `solidBodyMotionFvMesh`, that is what the `solidBodyMotionFunction` and derived classes

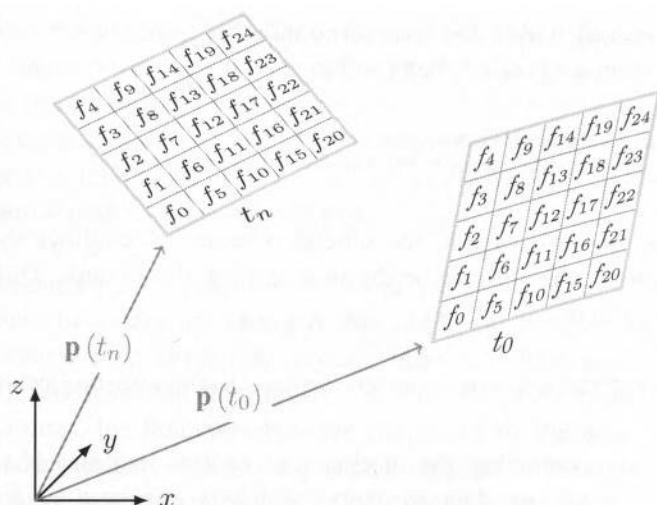


Figure 10.8: Illustration of an example patch moved from position at t_0 to the position at t_n with respect to the global coordinate system

are for,

- this simplifies separating the calculation of motion from the actual mesh motion algorithm,
- the `tabulatedSixDoF` class inherits from `solidBodyMotionFunction` which is the base class for all solid body motion functions in OpenFOAM.

The `solidBodyMotionFvMesh` is instantiated during construction of the `dynamicFvMesh` using runtime selection, if a dynamic mesh of type `solidBodyMotionFvMesh` is selected in the `dynamicMeshDict`. The particular parameters are read from a sub-dictionary of the `constant/dynamicMeshDict`, which is a private class attribute referred to as `SBMFCoeffs_`. The relevant parts of the program code for this boundary condition are described in the following text.

Reading the tabulated data from an input dictionary is shown in listing 87. The data and the filename are read from the sub-dictionary of the `dynamicMeshDict` into a list of type `Tuple2`. The `Tuple2` class is a data structure that stores two objects that can be of different type. In the above

Listing 87 Reading tabulated data from an input dictionary.

```

fileName newTimeDataFileName
(
    fileName(SBMFCoeffs_.lookup("timeDataFileName")).expand()
);
ifstream dataStream(timeDataFileName_);
List<Tuple2<scalar, translationRotationVectors> > timeValues
(
    dataStream
);

times_.setSize(timeValues.size());
values_.setSize(timeValues.size());

forAll(timeValues, i)
{
    times_[i] = timeValues[i].first();
    values_[i] = timeValues[i].second();
}

```

example, instances of a scalar and a translationRotationVectors are stored in the Tuple2, with the first one being the time and the second a nested vector, one for translation and one for rotation. Hence, the contents of the motion file are stored in that data structure directly. To provide easy access to that data, separate lists are created, as shown in the last half of the above code snippet.

Listing 88 Transformation member function of the tabulated motion boundary condition.

```

scalar t = time_.value();
// -- Some lines were spared --
translationRotationVectors TRV = interpolateSplineXY
(
    t,
    times_,
    values_
);

// Convert the rotational motion from deg to rad
TRV[1] *= pi/180.0;

quaternion R(TRV[1].x(), TRV[1].y(), TRV[1].z());
septernion TR(septernion(CofG_ + TRV[0])*R*septernion(-CofG_));

```

Calculation of the transformation from the input data is performed by the public member function `transformation()` and the relevant content is shown in listing 88. The second assignment interpolating the position and orientation data for the current time `t`. All angles must be converted to radians and finally a representation of the transformation is assembled, using quaternions and septernions.

Constructing an object is of course done by means of the constructor, that takes two arguments as shown in the listing below. The first one is a reference to the dictionary that contains the data required by `tabulated6DoFMotion`, which is the path to the data file. Passing a reference to `Time` simplifies the interpolation between the data points:

```
tabulated6DoFMotion
(
    const dictionary& SBMFCoeffs,
    const Time& runTime
);
```

After having found the code responsible for the point motion, the next step is to find a boundary condition that is derived from `pointPatchField` and performs a similar task than the one we plan to implement: from this we plan build our own boundary condition.

Adapting an Existing Boundary Condition

The already existing `oscillatingDisplacementPointPatchVectorField` boundary condition is a good starting point to derive the mesh motion boundary condition from. It applies displacement values according to a time dependent sinusoid to each of the *values* stored in the points on the boundary. The source code of `oscillatingDisplacementPointPatchVectorField` can be found in a subdirectory of `fvMotionSolver`:

```
?> $FOAM_SRC/fvMotionSolver/pointPatchFields/\
    derived/oscillatingDisplacement/
```

As discussed in the beginning of this chapter with regard to `fvPatchField`-type boundary conditions, the actual functionality of boundary conditions is implemented in either `evaluate()` or `updateCoeffs()` member functions. In case of the `oscillatingDisplacementPoint-`



`patchVectorField` boundary condition, it is the member function `updateCoeffs()` that calculates the displacement vector for each point of the boundary mesh. The displacement vector is defined like this:

```
amplitude_*sin(omega_*t.value())
```

with both `amplitude_` and `omega_` being scalar values (`omega` is the angular rotation) read from a dictionary. The implementation of this method can be performed as shown in listing 89.

Listing 89 Computing the displacement vector for the oscillating displacement boundary condition.

```
void oscillatingDisplacementPointPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {
        const polyMesh& mesh =
            this->dimensionedInternalField().mesh();

        const Time& t = mesh.time();

        Field<vector>::operator=(amplitude_*sin(omega_*t.value()));

        fixedValuePointPatchField<vector>::updateCoeffs();
    }
    fixedValuePointPatchField<vector>::updateCoeffs();
}
```

The line

```
Field<vector>::operator=(amplitude_*sin(omega_*t.value()));
```

of the `updateCoeffs()` method uses the assignment operator of `Field` to assign the proper displacement values to the boundary points of the mesh. This instruction is followed by a call to `updateCoeffs()` of the parent class. By assigning the displacement of the patch to the patch points, the dynamic mesh solver takes care of the actual mesh motion.

Assembling the Boundary Condition

A working version of this boundary condition can be found in the example code repository distributed along with the book, bundled together with the

recirculation control boundary condition of the previous chapter into the library `primerBoundaryConditions`. For easier understanding, you may want to have the source code of the ready-to-use mesh motion boundary condition open in your text editor while following the steps described here. Similar to other programming examples the boundary condition of this example should be compiled into a dynamic library using `wmake libso`. As usual the first step is to create a new directory to store the boundary condition in:

```
?> mkdir -p \
    $WM_PROJECT_USER_DIR/applications/tabulatedRigidBodyDisplacement
?> cd $WM_PROJECT_USER_DIR/applications/tabulatedRigidBodyDisplacement
```

The next step is to copy the `oscillatingDisplacementPointPatchVectorField` to the new directory:

```
?> cp $FOAM_SRC/fvMotionSolver/pointPatchFields/\
    derived/oscillatingDisplacement/* .
```

To keep the `tabulatedRigidBodyDisplacement` boundary condition named properly, all occurrences of `oscillatingDisplacement` must be replaced by `tabulatedRigidBodyDisplacement`. This applies both for the filenames as well as matches inside the source files themselves. After deleting `*.dep` files the remaining C and H files have to be renamed accordingly.

```
?> rm *.dep
?> mv oscillatingDisplacementPointPatchVectorField.H \
    tabulatedRigidBodyDisplacement.H
?> mv oscillatingDisplacementPointPatchVectorField.C \
    tabulatedRigidBodyDisplacement.C
?> sed -i "s/oscillating/tabulatedRigidBody/g" *.H
```

The first thing to check is if the renamed `oscillatingDisplacementPointPatchVectorField` still compiles properly. To check this the OpenFOAM typical `Make/files` and `Make/options` files need to be created:

```
?> mkdir Make
?> touch Make/files
?> touch Make/options
```



The content of files is short, as the boundary condition consists of only one source file:

```
tabulatedRigidBodyDisplacementPointPatchVectorField.C

LIB = $(FOAM_USER_LIBBIN)/libtabulatedRigidBodyDisplacement
```

For sake of simplicity, all example boundary conditions provided in the code repository are compiled into one library. This library's name differs from the one defined in the above code snippet.

This source file has plenty of dependencies, though. Various other libraries and their header files have to be linked to this library:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/fileFormats/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -lfileFormats
```

Test if the boundary condition still compiles, by issuing a `wmake libso` in the directory that contains the `Make` folder. If everything compiles without errors and warnings, open both the header and the source file in an editor of your choice and apply the following changes. First, header files have to be included in `tabulatedRigidBodyDisplacementPointPatchVectorField.H`:

```
#include "fixedValuePointPatchField.H"
#include "solidBodyMotionFunction.H"
```

And the source file must include some more header files:

```
#include "tabulatedRigidBodyDisplacementPointPatchVectorField.H"
#include "pointPatchFields.H"
#include "addToRunTimeSelectionTable.H"
#include "Time.H"
#include "fvMesh.H"
```

```
#include "Ifstream.H"
#include "transformField.H"
```

The actual implementation of the desired functionality is done in the `updateCoeffs()` member function. Although one private attribute is used there that is not implemented, yet: a constant dictionary that contains all data defined in the boundary condition's dictionary in the `0/` directory. This dictionary is added to the private attributes in the header file:

```
// - Store the contents of the boundary condition's dictionary
const dictionary dict_;
```

Each of the boundary condition's constructors must initialize the new private attribute, which is usually done by calling the null-constructor of dictionary. An example is the constructor that constructs the boundary condition from a `pointPatch` and a `DimensionedField`, and it is shown in listing 90. In case the boundary condition is constructed by reading the

Listing 90 Constructor of the point displacement boundary condition.

```
tabulatedRigidBodyDisplacementPointPatchVectorField::
tabulatedRigidBodyDisplacementPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(p, iF),
    dict_()
{}
```

particular file from the `0` directory, the following constructor is called. As a matter of fact, the dictionary of the boundary condition is passed to the constructor and it must be stored in the boundary condition for later processing:

```
tabulatedRigidBodyDisplacementPointPatchVectorField::
tabulatedRigidBodyDisplacementPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
{}
```



```

:
  fixedValuePointPatchField<vector>(p, iF, dict),
  dict_(dict)
{
  updateCoeffs();
}

```

This constructor is the only one that calls `updateCoeffs()` during construction. The `updateCoeffs` member function is shown in listing 91

Listing 91 The `updateCoeffs` member function of the tabulated rigid body motion boundary condition.

```

void tabulatedRigidBodyDisplacementPointPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const polyMesh& mesh = this->dimensionedInternalField().mesh();
    const Time& t = mesh.time();
    const pointPatch& ptPatch = this->patch();

    autoPtr<solidBodyMotionFunction> SBMFPtr
    (
        solidBodyMotionFunction::New(dict_, t)
    );

    pointField vectorIO(mesh.points().size(), vector::zero);

    vectorIO = transform
    (
        SBMFPtr().transformation(),
        ptPatch.localPoints()
    );

    Field<vector>::operator=
    (
        vectorIO-ptPatch.localPoints()
    );

    fixedValuePointPatchField<vector>::updateCoeffs();
}

```

The most important line is the line that defines the `SBMFPtr`. It constructs an `autoPtr` for a `solidBodyMotionFunction`, based on a dictionary and an object of `Time`. As this code originates from the `dynamicFvMesh` library, the dictionary passed to the constructor is a subdictionary of

the `dynamicMeshDict`. This subdictionary contains all the parameters required by the `solidBodyMotionFunction` selected by the user in the `dynamicMeshDict`. As the definition of the motion parameters for this boundary condition should be performed on a per-boundary basis and not a global-basis, the dictionary that is passed to the constructor should be read from the boundary condition in the 0 directory, rather than the `dynamicMeshDict`. This is what the private member `dict_` is for: It is read only once and passed to the `solidBodyMotionFunction` in each call to `updateCoeffs()`.

The next lines are similar to the ones that can be found in the `tabulated6DoFMotion`. The absolute positions of the patch points after the transformation is applied is stored in `vectorIO`. As the actual motion is relative to the previous point positions, the difference must be computed which is done directly in the call to the assignment operator. The transformation is performed using septernions, which are superior to transformation matrices in many regards.

Now that the source code has been prepared, compile the library again.

```
?> wclean
?> wmake libso
```

Executing the Simulation in the Example Case

The execution of the simulation in the example case should be done at first with the prepared and tested code from the example code repository. Once the distributed code is run, the necessary input parameters in the dictionary, as well as the modifications applied to the fields should be clear and the implemented boundary condition can be tested. The example case `tabulatedMotionObject` is located in the example case repository. This three dimensional case is a simple demonstration of the functionality of the newly implemented tabular rigid body motion boundary condition. Figure 10.9 shows a sketch of the case setup, including a cubic domain with a volume cut out of the middle. The boundary created by this cutting process is represented by the `movingObject` patch.

The `movingObject` boundary will be displaced by the new boundary condition according to the data contained in the `6DoF.dat` file in `con-`



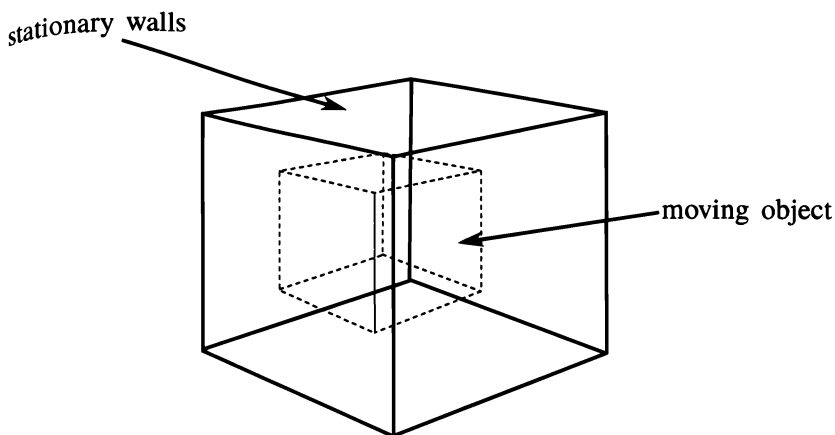


Figure 10.9: Illustration of the domain and moving patch of the tabulatedMotionObject example case.

stant/. Compared to a basic OpenFOAM case, there are new configuration files that are required to execute the new boundary condition: The 0/pointDisplacement and constant/dynamicMeshDict files. Initial boundary values are set for the movingObject patch of the pointDisplacement field using the new boundary condition, as shown in listing 92. In order to use the boundary condition, the instruction to dy-

Listing 92 The movingObject dictionary.

```
movingObject
{
    type            tabulatedRigidBodyDisplacement;
    value           uniform (0 0 0);
    solidBodyMotionFunction tabulated6DoFMotion;
    tabulated6DoFMotionCoeffs
    {
        CofG        ( 0 0 0 );
        timeDataFileName "constant/6DoF.dat";
    }
}
```

ynamically load the new library at runtime needs to be added to the system/controlDict configuration file. Please note, if you used the code presented in this section to compile the boundary condition, the library is named differently ("libprimerBoundaryConditions.so").

The `dynamicMeshDict` is needed to control both the mesh motion solver. For this case the `displacementLaplacian` smoother is used, which is based on Laplace's equation and smooths the point displacements. If the surrounding points did not move with the patch, proximal cell would be quickly deformed and crushed. The contents of the `dynamicMeshDict` are shown in listing 93.

Listing 93 The `dynamicMeshDict` for the tabulated motion example.

```
dynamicFvMesh      dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver             displacementLaplacian;

displacementLaplacianCoeffs
{
    diffusivity      inverseDistance (floatingObject);
}
}
```

The simulation can be run by executing the `Allrun` script within the simulation case directory. This script performs all necessary steps, such as mesh generation, to complete this example hassle-free. The utility solver `moveDynamicMesh` only calls dynamic mesh routines and is devoid of any flow related computations, which makes it relatively fast compared to the usual flow solvers with dynamic mesh capabilities. In addition to performing mesh motion operations it executes numerous mesh related quality checks.

Post-processing the case is straightforward and can be performed visually: the case can be inspected in Paraview. Animating the case shows the internal patch tumbling and swaying around within the domain, similar to the data provided in the input file. Cutting the domain in half using the `Clip` filter with the option `Crickle Clip`² clarifies some interesting things with regards to dynamic meshes: The way the point displacement imposed by the boundary condition is dissipated by the mesh motion solver. The Laplace equation dissipates the displacement to internal mesh points, moves the points accordingly and maintains good cell quality while the patch translates and rotates. Figure 10.10 holds the image of

²Available in ParaView version 4.0.0 or higher.



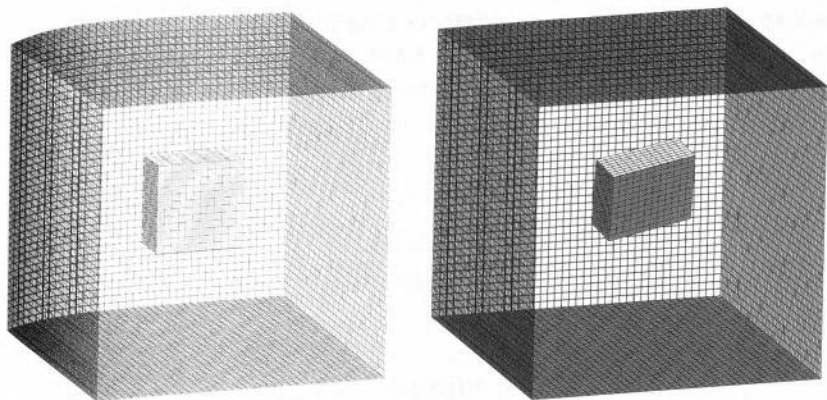


Figure 10.10: Illustration `tabulatedMotionObject` case before and after patch transformation.

the final patch position before and after transformation.

Summary

In this chapter the design and implementation of boundary conditions for the FV method and mesh motion in OpenFOAM was described. Both families of boundary conditions are modelled as two separated class hierarchies, `fvPatchField` and `pointPatchField`, respectively. Dynamic polymorphism allows different extensions and combinations of existing implementations. Having boundary conditions implemented as a class hierarchy in conjunction with the fields stored in the internal mesh points allows OpenFOAM library to determine the type of boundary condition during runtime, without recompiling the code each time a new boundary condition is set for a field. The mechanism for loading of dynamic libraries allows new implementations of boundary conditions being compiled into separate libraries (to e.g. the `finiteVolume` library). This in turn results in a straightforward way to add new boundary conditions, without having to re-compile the numerical library. These advantages allow the programmer to extend the boundary conditions, and other parts of OpenFOAM, by developing (and sharing) self-sustained libraries.

Developing new boundary conditions involves finding a point of entry in the class hierarchy, from which hierarchy can be extended. The choice of

class as a starting point for further developments for a boundary condition depends on the functionality the desired boundary condition implements, and if a similar boundary condition already exists. The computational part the boundary conditions is located in the `updateCoeffs` member function. Apart from renaming the source files appropriately and adapting the new class for inheritance, the programmer will most likely do most of his work implementing the body of this function.

Further reading

Gamma, Erich et al. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2.

OpenFOAM User Guide (2013). OpenCFD limited.



11

Transport Models

The transport model library in OpenFOAM can be divided into two major class types and their derivatives: transport models and viscosity models. None of them actually transport any properties directly as the transport of properties is computed by the overall CFD solution algorithm. Rather than transporting flow properties, viscosity models are responsible for modeling the kinematic viscosity ν , which depends on the selected fluid type, such as Newtonian and non-Newtonian fluid and local flow conditions. Transport models provide access to the viscosity models in the top-level code. Depending on the flow type (single phase or multi-phase), different transport models need to be selected.

This chapter provides a brief introduction of the physical effects that relate to viscosity in one way or another. The design of the class interface outlined in the previous paragraph is described and a detailed description is provided on how to implement a new viscosity model in OpenFOAM.

11.1 Numerical Background

Models for a variety of fluid types are already shipped with the official OpenFOAM release. A brief introduction to the physical and numerical aspects of the viscosity models is provided in this section. More information on the actual implementation and on the class interface can be found

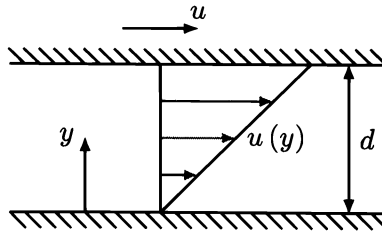


Figure 11.1: Couette flow with velocity distribution

in section 11.2. For a more detailed description of the physics behind viscosity, please refer to Schlichting and Gersten 2001 and other fluid dynamics text books such as Wilcox 2007.

A flow configuration often used for describing the viscosity of fluids is the so-called *Couette flow*. It is described as the flow between two parallel plates with large spacial extend and the distance d between them clarifies the effects of viscosity in a straight forward way. The lower plane is fixed in space and does not move, whereas the upper plane moves constantly at a velocity \mathbf{u} .

A velocity gradient between both plates develops over time, as outlined in figure 11.1. This results in a shear stress τ acting upon the upper plate that Schlichting and Gersten 2001 defines as

$$\tau = \mu \frac{du}{dy}, \quad (11.1)$$

with μ denoting the dynamic viscosity that relates to the *kinematic* viscosity ν in the following manner:

$$\nu = \frac{\mu}{\rho}. \quad (11.2)$$

The kinematic viscosity ν is the property that is usually specified in OpenFOAM. While most fluids can be described as Newtonian, certain



fluids exhibit non-linear stress-strain relationships. These special materials demand specific models for the viscosity itself. Luckily, OpenFOAM includes various viscosity models along with the standard Newtonian model:

Newtonian for incompressible Newtonian fluids with $\nu = \text{const}$,
BirdCarreau for incompressible Bird-Carreau non-Newtonian fluids,
CrossPowerLaw for incompressible Cross-Power law that is based non-Newtonian fluids,
HerschelBulkley for Herschel-Bulkley non-Newtonian fluids,
powerLaw for power-law based non-Newtonian fluids.

The viscosity model is not the only model that affects the viscosity, however. The selected turbulence model changes the *effective* viscosity which is used by the solver (chapter 7). If turbulence modeling is disabled and the laminar turbulence model is selected, ν remains unaltered.

11.2 Software Design

As described in the introduction of this chapter, the library which contains the transport models of OpenFOAM can be subdivided into two major components: transport models and viscosity models. Compared to many other classes in the OpenFOAM release, the majority of classes in the transport model library do not perform sophisticated operations and end the end that is not their purpose. Instead, they rather provide an easy and object oriented access to viscosity related data. The following description involves various classes which, while they strongly relate to each other, their major purpose is to structure data logically. A simplified UML diagram for the major classes is provided in figure 11.2 and can be regarded as a visualization of this section.

From a user's point of view, both of the above mentioned categories for transport models can be combined into one general construct. For a programmer, on the other hand, the reasons why this separation is used emerge after studying the source code. Here there are two transport models and one base class. The base class is called `transportModel` and inherits from `IObject`. The `transportModel` class is initialized in the constructor to read the `constant/transportProperties` dictionary

of the respective OpenFOAM case. Listing 94 shows the constructor of `transportModel` and the initialization of the `IObject`:

Listing 94 Constructor of `transportModel`

```
Foam::transportModel::transportModel
(
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    IOdictionary
    (
        IObject
        (
            "transportProperties",
            U.time().constant(),
            U.db(),
            IObject::MUST_READ_IF_MODIFIED,
            IObject::NO_WRITE
        )
    )
{}

```

The available two implementations of transport models differ only between single and two-phase models, which compose the two derivatives of the `transportModel` class: `singlePhaseTransportModel` and `incompressibleTwoPhaseMixture`. This division is due to the natural differences of physical phenomenon present in two-phase and single-phase flows. Each phase of a two-phase flow, for example, may possess drastically different physical properties.

Before reviewing the two-phase transport model, the single-phase flow model is described in the following. In OpenFOAM, all single phase solvers employ the `singlePhaseTransportModel` class to gain access and load the kinematic viscosity `nu`. This parameter must be provided by the user in the `constant/transportProperties` dictionary. This is very convenient, as the `singlePhaseTransportModel` and other transport models delegate the process of reading and updating this dictionary which helps keep the code clean. Within the single phase solvers, the `singlePhaseTransportModel` is instantiated as follows:

```
singlePhaseTransportModel laminarTransport(U, phi);
```

Note that the `laminarTransport` object is later passed to the constructor



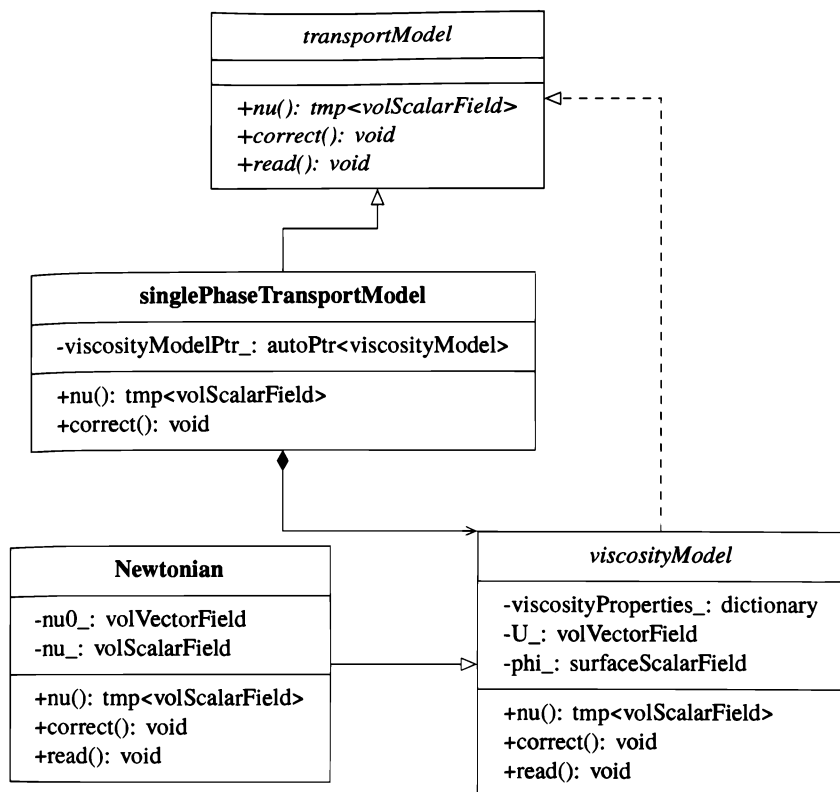


Figure 11.2: Class collaboration diagram for the transport models.

of the turbulence model. This is due to the fact that a turbulence model will require access to the *laminar* viscosity, that in turn is described by the `singlePhaseTransportModel`.

Both, `transportModel` and `singlePhaseTransportModel` define a public member function `nu()` which returns the viscosity as a `volScalarField`. Because `transportModel` is an abstract base class, it does not implement this member function whereas the `singlePhaseTransportModel` class must implement this member function (see figure 11.2). Its implementation, however, delegates the functionality to the `viscosityModel`:

```

Foam::tmp<Foam::volScalarField>
Foam::singlePhaseTransportModel::nu() const

```

```
{
    return viscosityModelPtr_ -> nu();
}
```

The `viscosityModelPtr_` is a private member of `singlePhaseTransportModel` and is defined as an `autoPtr` to a `viscosityModel`. This class attribute is defined in the header file as:

```
autoPtr<viscosityModel> viscosityModelPtr_;
```

For now, it is only important that the `viscosityModel` somehow determines the kinematic viscosity. Its actual implementation and how the `viscosityModel` is selected are both discussed later in this section.

The two phase transport model is modelled by the `incompressibleTwoPhaseMixture` class, used by the `interFoam`-type solvers in OpenFOAM that do not involve phase change. In a similar manner to most single phase solvers, `interFoam` type solvers instantiate an object of that class in the following way:

```
incompressibleTwoPhaseMixture twoPhaseProperties(U, phi);
```

The class collaboration for the incompressible two phase mixture model is shown in figure 11.3. This instantiation is similar to the approach that has been used for the `singlePhaseTransportModel`, however, the `incompressibleTwoPhaseMixture` class stores additional data. Rather than having one `viscosityModel`, two are instantiated and stored by that class to account for each fluid phase.

The class definition of `incompressibleTwoPhaseMixture` is shown in listing 95. Using an `autoPtr` to store objects of any derived classes of `viscosityModel` is mandatory, because there are several different classes for different fluid types, that are derived from `viscosityModel`. Each of them is run time selectable and finally determines what kind of fluid is selected which in turn defines the viscosity and hence the return value of the public member function `nu`. To differentiate between both fluid phases, a new `volScalarField` is introduced: `alpha1_`. This field is used in the `VoF` member function and is effectively a blending value between both phases' properties such as viscosity and density. Unlike the



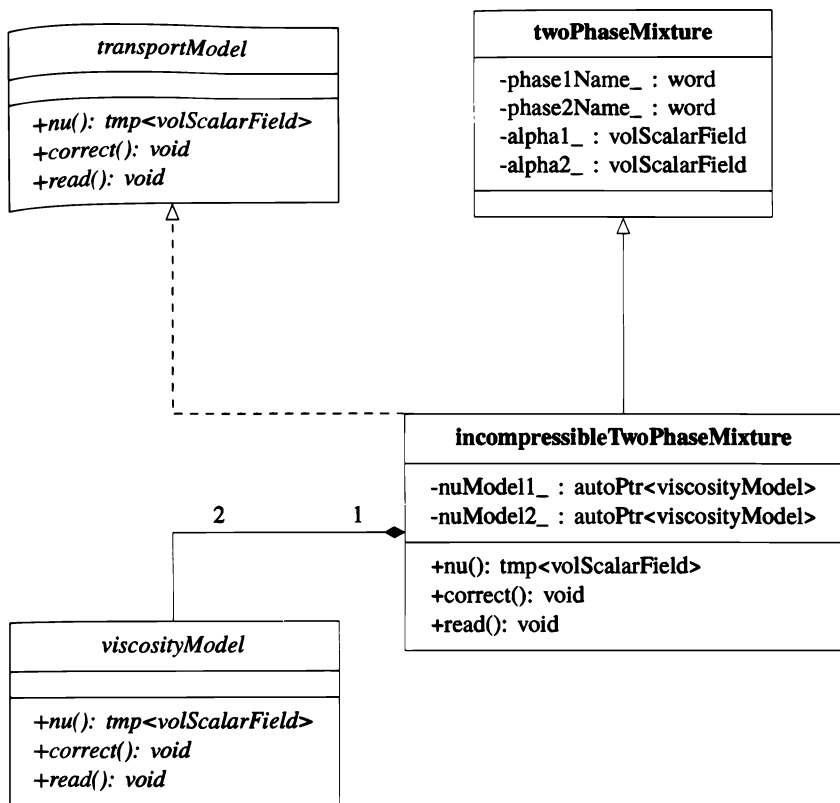


Figure 11.3: Class collaboration diagram for the incompressible two phase mixture model.

`singlePhaseTransportModel`, where a call to the public member function `nu` is delegated directly to the chosen `viscosityModel`, the return value of this public member function is composed differently. A copy of the private member `nu_` is returned, which is calculated based on both phases' `viscosityModels` and the current `alpha1_` field. This calculation is performed by the private member `calcNu` and triggered when the public member function `correct` is called as shown in listing 96.

As seen in the sources, both `viscosityModels` are updated and the volume fraction `alpha1_` is limited to be bounded between 0 and 1. Finally, the kinematic viscosity is calculated using the dynamic viscosity `mu` and density distribution. The latter is computed using the rule of mixture in

Listing 95 Class definition of `incompressibleTwoPhaseMixture`

```
class incompressibleTwoPhaseMixture
:
    public transportModel,
    public twoPhaseMixture
{
protected:

    // Protected data

    autoPtr<viscosityModel> nuModel1_;
    autoPtr<viscosityModel> nuModel2_;

    dimensionedScalar rho1_;
    dimensionedScalar rho2_;

    const volVectorField& U_;
    const surfaceScalarField& phi_;

    volScalarField nu_;
```

terms on each phase's density and the volume fraction `alpha1_`. The dynamic viscosity `mu` is calculated similarly, using a bounded `alpha1_` field and the kinematic viscosities `nu` of each `viscosityModel`.

Until now, the `transportModel` and both of its derivatives were focused on and the `viscosityModels` themselves were mentioned only briefly. The `viscosityModels` are those classes which are responsible for modelling and computing the viscosity, following the OOD pattern Single Responsibility Principle (SRP). A list of viscosity models and their description can be found in section 11.1. Similar to the structure of `transportModel`, the `viscosityModel` class is an abstract base class for the actual viscosity models.

As outlined in figure 11.2, the `viscosityModel` implements a public member function `nu`, which finally returns the kinematic viscosity. This member function is the one accessed in any of the `transportModels` to access the viscosity. In the base class `viscosityModel`, this is a virtual member function and needs to be implemented by each derived class:

```
virtual tmp<volScalarField> nu() const = 0;
```

With `singlePhaseTransportModel` and `incompressibleTwoPhaseMixture`



Listing 96 Calculation of cell centered viscosity for a two phase mixture.

```

void Foam::incompressibleTwoPhaseMixture::calcNu()
{
    nuModel1->correct();
    nuModel2->correct();

    const volScalarField limitedAlpha1
    (
        "limitedAlpha1",
        min(max(alpha1_, scalar(0)), scalar(1))
    );

    // Average kinematic viscosity calculated from dynamic viscosity
    nu_ = mu()/(limitedAlpha1*rho1_ + (scalar(1) - limitedAlpha1)*rho2_);
}

```

ture being hard coded into the particular solver, the `viscosityModels` are the final types which are selected by the user. They hence have to be run time selectable and be selected by specific entries in the `constant/transportProperties` dictionary. Because of this, the base class must implement the OpenFOAM RTS mechanism.

As an example for a `viscosityModel` the `Newtonian` class is selected. It describes the viscosity for an incompressible Newtonian fluid and solely inherits from `viscosityModel`. Additional data is stored in the following private member variables:

```

dimensionedScalar nu0_;

volScalarField nu_;

```

The `nu` member function must be implemented by this class, as its base class definition is virtual. This implementation is short and simply returns `nu_`. To clarify the construction of this class, the constructor is shown in listing 97.

Of course the base class' constructor is called at the first position of the initialization list, followed by initializations for `nu0_` and `nu`.

Listing 97 Constructor of the Newtonian class

```

Foam::viscosityModels::Newtonian::Newtonian
(
    const word& name,
    const dictionary& viscosityProperties,
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    viscosityModel(name, viscosityProperties, U, phi),
    nu0_(viscosityProperties_.lookup("nu")),
    nu_
    (
        IOobject
        (
            name,
            U_.time().timeName(),
            U_.db(),
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        U_.mesh(),
        nu0_
    )
{}

```

TIP

Even though the `transportModel` is the abstract base class for all transport models, specialistic solvers will select other model classes in the hierarchy. An example is the `incompressibleTwoPhaseMixture`, which *is a* `transportModel`, but is also a `twoPhaseMixture`, specifically meant to be used by the two-phase solvers.

11.3 Implementation of a new Viscosity Model

To clarify the implementation of a custom `viscosityModel`, a model based on raw rheometry data will be used as an example. This rheometry data is stored in a tabulated text file with the first column being the strain rate and the second the corresponding effective dynamic viscosity μ . We will generate a new viscosity model class which takes an inputs of the data table and the local strain-rate and returns the local effective viscosity. This data table based approach is in contrast to the other more common viscosity models based on analytical expressions of the strain-



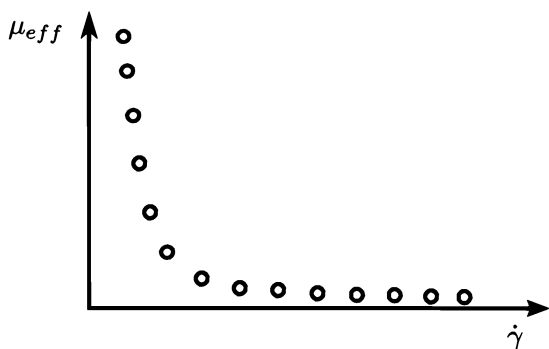


Figure 11.4: An example of a measured effective viscosity profile for a non-Newtonian fluid.

rate vs. viscosity relationship. An example of a set of measurements from a shear rheometer (effective viscosity vs. strain rate) is shown in figure 11.4. Because there are only discrete data points in the rheometry table, an interpolation scheme must be used to interpolate between them and assign a viscosity to any given strain rate. OpenFOAM ships with a two dimensional spline based interpolation which can be used for this purpose: `interpolateSplineXY`. The source code for this library is included in the example repository.

Passing the rheology data to the new viscosity model is done by selecting the new viscosity model in `constant/transportProperties` and adding a sub-dictionary to it:

```
transportModel interpolatedSplineViscosityModel;

interpolatedSplineViscosityModelCoeffs
{
    dataFileName    "rheologyTable.dat";
}
```

This sub-dictionary contains the name of the file which contains the tabulated data (assumed to be stored in the `constant` directory). The spline

interpolation method is called in the private member function `calcNu` where a bulk of the class activity occurs (see listing 98).

The code functions by first executing the `loadDataTable()` private member function. This function opens and parses the plain text data table into the `rheologyTableX_` and `rheologyTableY_` members. After initializing a `viscosityField` and creating a pointer to the strain-rate field, we loop through all cells and compute a local effective viscosity based on a fitted spline. The `interpolateSplineXY` class requires three inputs to compute the interpolated value: the local shear-rate magnitude for input, and the `x` and `y` lists which represent the 2D data plot. With an effective viscosity computed for each cell in the domain, it is returned for use in the diffusive term of the momentum equation.

11.3.1 Example Case

A simple example case illustrating the usage of the spline-interpolation based viscosity model is chosen. The case itself is included in the example cases repository and is entitled `nonNewtonianDroplet`.

A two-phase VoF solver `interFoam` is used to simulate a non-Newtonian liquid droplet falling through air and impacting a solid surface below. Note that the simulation is two dimensional and is spatially under-resolved, however, it serves well for the example simulation case. The square domain is bounded by three solid walls and one open atmosphere boundary condition as shown in figure 11.5. The case can be run to completion using the prepared `Allrun` script.

The simulation begins in a quiescent state with the droplet centered in the domain. Gravity accelerates the droplet downward where it eventually impacts the solid, no-slip wall. Once the droplet begins to impact the wall, high local strain rates rapidly alter the effective viscosity in the liquid. The non-Newtonian effects can be visualized by observing the change in the `nu1` field during impact as shown in figure 11.6.

Looking at the time lapse images, one can see that the non-Newtonian viscosity is being calculated for all cells in the domain regardless of the phase: the gas phase of the simulation represents air, a fluid with Newtonian rheology. In this case the non-Newtonian viscosity is only



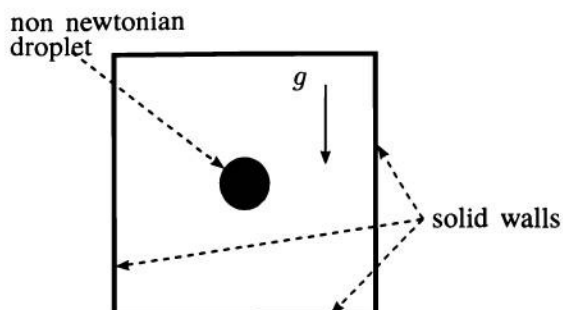


Figure 11.5: The initial domain for the nonNewtonainDroplet example case.

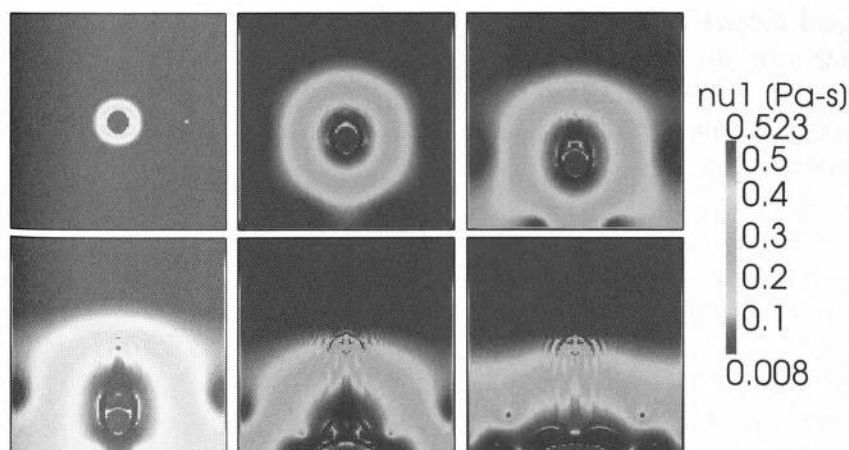


Figure 11.6: A time lapse of the decent and impact of the non-Newtonian droplet.

being applied to the liquid phase of the simulation due to the VoF approach to material property weighting. The rapid change of the droplet's internal viscosity is clearly visible, as it shears against the bottom wall of the domain. The effective viscosity then increases as the droplet comes to rest and the local strain-rate approaches zero.



Listing 98 Interpolation of viscosity by calcNu

```

Foam::volScalarField
Foam::viscosityModels::interpolatedSplineViscosityModel::calcNu()
{
    //Open and parse the rheometry data file
    loadDataTable();

    //Initialize a viscosity field
    volScalarField viscosityField
    (
        IOobject
        (
            "viscosityField",
            U_.time().timeName(),
            U_.db(),
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        U_.mesh(),
        dimensionedScalar("viscosityField", dimensionSet(0,2,-1,0,0,0,0), 0.0),
        zeroGradientFvPatchScalarField::typeName
    );

    //Access the local strain rate
    const volScalarField& localStrainRate = strainRate();

    //Loop through all cells and calculate the effective viscosity
    // using the spline interpolation class
    forAll(localStrainRate.internalField(), cellI)
    {
        scalar& localMu = viscosityField.internalField()[cellI];

        localMu = interpolateSplineXY
        (
            localStrainRate.internalField()[cellI],
            rheologyTableX_,
            rheologyTableY_
        );
    }

    return viscosityField;
}

```

Further reading

- Schlichting, Hermann and Klaus Gersten (2001). *Boundary-Layer Theory*. 8rd rev. ed. Berlin: Springer.
- Wilcox, D. C. (2007). *Basic Fluid Mechanics*. 3rd rev. ed. DCW Industries Inc.



12

Function Objects

A function object is a solver independent code that is compiled into a dynamic library and executed during the time loop of a solver or utility, to perform various tasks and calculations. Usually it serves the purpose of performing runtime field post-processing, but it can also be used to manipulate the fields and the simulation parameters during run time. Just like any other C++ library, a single function object library can contain an arbitrary number of function objects and will be linked to the application during runtime. The encapsulated functionality of the function objects is accessed from the solver in a predefined way, so the function objects must all adhere to a specified fixed class interface.

These aspects make function objects very similar to transport models and boundary conditions: both transport models and boundary conditions conform to different class interfaces, defined by their respective base class. They are compiled into separate libraries and dynamically loaded and linked to the solver. None of them are hard-linked with the solver, nor must they be recompiled with the solver. This allows the user to select function objects via configuration files, even during runtime. An other benefit is that the libraries and their particular source code can be shared easily with others.

Both transport models and boundary conditions manipulate the field values of the simulation as parts of the solution algorithm of the simulation.

In comparison, function objects operate in a very different way: they implement fully self sustained operations from the solution algorithm and do not involve any modification of the application code at all. No changes must be applied to the solvers or utility applications and there is no need to recompile them in order to access new function objects, either. This improves the re-usability of the code following the OOD pattern of "Open-Closed": functionality is open for extension and the existing implementation is closed for modification.

From a user's perspective, function objects usually provide functionalities that do not affect the solution. They rather perform runtime post-processing tasks, that should generally be independent of the selected solver. Their purpose is to separate general post-processing methods, such as calculating average values of any field variable from the solver code. This is supposed to be done in such a way that the calculation is performed during the simulation.

As a hypothetical example, consider a parametric CFD optimization study with a pressure drop between the inlet and outlet boundary as the target function. Using function objects, it is possible to evaluate this pressure drop after each time step during the simulation and terminate the simulation when the pressure drop suffices some desired conditions. Then the optimization parameters must be modified and the optimization cycle is to be continued afterwards. Post-processing applications would expect the simulation to be completed, which would involve a full run of every simulation case in the optimization parameter space, which in turn would incur more computational costs.

12.1 Software Design

This section covers two aspects of function object design: function objects as they are implemented in C++ directly and those OpenFOAM ships with. Both do a somewhat similar job, but their implementation differs in some regards.

The `rising-bubble-2D` case of the case repository serves as an example case for this section. The aim is to implement a function object in OpenFOAM, that selects cells based on the value of a scalar field. To il-



illustrate the working principles and differences between both function object types, this functionality is implemented using both types of function objects: C++ Standard Template Library (STL) and OpenFOAM function objects.

12.1.1 Function Objects in C++

Describing the implementation and usage of function objects in C++ is beyond the scope of this book and described extensively in Josuttis 1999, Vandevoorde and Josuttis 2002 and Alexandrescu 2001, among others. In addition to that, there is an ample amount of free information describing function objects available on the internet, so only a brief overview of function objects is provided in this section. It is sufficient for understanding the basis of the implementation of function objects in OpenFOAM and some differences between the OpenFOAM implementation and the common implementation of function objects in the C++ language.

As their name indicates, function objects are objects that can behave like functions. In the C++ programming language, an object can behave like a function when the operator `operator()` is overloaded for its class. A very simplified form of the implementation of such a class is shown in listing 99. Overloading arithmetic operators (`+`, `-`, `*`, `/`) allows the class

Listing 99 Simplified form of a C++ function object implementation

```
class CallableClass
{
    public:

        void operator()() {}
};

int main(int argc, const char *argv[])
{
    CallableClass c;

    c();

    return 0;
}
```

to implement arithmetic operations on objects, though the implementation is not limited to arithmetic operations. This C++ language feature is used extensively for algebraic field operations in OpenFOAM itself. In a

similar way, overloading `operator()` for a class is commonly used to enable function-like behaviour of its objects.

The fact that function objects are objects comes with multiple advantages:

- The function object can store additional information as class attributes
- It is implemented as a type which is a fact often used by generic programming design
- Its execution will be faster than the code which involves passing a function pointer (function objects are often inlined)

A function object can accumulate information while processing arguments in the `operator()` and store the accumulated information in the class attributes for later use. Storing that data can be done outside of the scope where it is called from. This is possible because the scope of the function implemented in the overloaded `operator()` is linked to the scope of the function object by implementing the functionality within the class `operator()`.

The following example of C++ function objects in action shows how to implement an OpenFOAM class that selects mesh cells with the help of the C++ STL function objects, instead of relying heavily on existing OpenFOAM data structures. This class is named `fieldCellSet` and is available in the example code repository, within the `primerFunctionObjects` sub-directory.

Cell selection in OpenFOAM is usually done using a topological selection: e.g. selecting cells whose centers lie within a sphere. The `fieldCellSet` class implemented in this example uses field values of a `volScalarField` to collect cells from the mesh. If the field values satisfy a specific criterion, the cell selector will add the particular cell label to a set of labels.

Class `fieldCellSet` is decoupled from the cell selection criterion, following the SRP, by defining the cell selection function as a function template parametrized on the selection criterion. In the following two code snippets, this template parameter is of generic type `Collector` and



the parameter `col` stores an object of this type. Hence `fieldCellSet` uses template arguments to describe the actual selection criterion in `collectCells`.

```
//- Edit
template<typename Collector>
void collectCells(const volScalarField& field, Collector col);
```

The ability of `fieldCellSet` to work with any `Collector` function object that has the `operator()` implemented, takes a scalar value and returns a boolean is a result of this. The concept of the `Collector` template parameter can be examined in the implementation of the `collectCells` member function template, defined in file `fieldCellSet-Templates.C`:

```
template<typename Collector>
void fieldCellSet::collectCells
(
    const volScalarField& field,
    Collector col
)
{
    forAll(field, I)
    {
        if (col(field[I]))
        {
            insert(I);
        }
    }
}
```

As the above snippet shows, the `collectCells` member function simply iterates over all field values and expects `Collector` to return a boolean variable as a result of operating on the field value `field[I]`. This specifies the collector template parameter concept:

callable: a function object is the natural choice

unary: it must allow at least one function argument

predicate: it returns a boolean variable

In summary, the small cell collector class `fieldCellSet` has a member function template that takes a `volScalarField` and a function object `col` as arguments. It uses the function object to check if the cell

should be added to the cell set, based on the value of the field. However, it is irrelevant how this comparison is implemented, as long as the `operator()` of `col` is implemented and returns a boolean.

This straightforward implementation allows *any* function object to be passed to the `collectCells` member function template. In the following example, a function object of the STL is used. The implementation of the test application for a single `fieldCellSet` class is available in the example code repository, under the `applications/test` sub-directory and it is named `testFieldCellSet`. The part of the `testFieldCellSet` that relates to function objects in C++ is a single line of code:

```
fcs.collectCells(field, std::bind1st(std::equal_to<scalar>(), 1));
```

The `collectCells` method is called on the `fcs` object of the `fieldCellSet` class.

The function object used in this example is the `equal_to` STL function object template. This generic function object simply checks whether two values of type `T` are identical. Since it uses *type lifting* aspect of generic programming, it may only be applied on instances of types that have the equality operator `==` defined. However, the `equal_to` function object requires two arguments for the comparison. The question that remains is how it can be called for the field value within the `collectCells` member function template. For this simple example, the answer is fairly simple: The first argument of the `equal_to` function object has been bound to the value of 1. Meaning that the `testFieldCellSet` application should create a cell set consisted of all mesh cells with the field values equal to 1.

Object oriented design allows the `fieldCellSet` class to delegate the storage of the cell labels as well as delegate the output operation which is performed at each new time step. This was achieved using multiple inheritance:

```
class fieldCellSet
:
    public labelHashSet,
    public regIOobject
{
```



In the above snippet, the `labelHashSet` is an instantiation of the `OpenFOAM` class template `HashSet` using `Foam::label` as key values. Inheriting from `regIOobject` allows the class to be registered in an object registry and written to disk automatically as time increment operator (`Time::operator++()`) is invoked. The writing is performed in such a way that the output file contains the necessary OpenFOAM file header information required by OpenFOAM to reread the data later on.

Testing the functionality of this example's application can be done quite straightforwardly, using the test case described previously. If the application is invoked on the `alpha1` field of the `rising-bubble-2D` example case, it will extract the cells of the bubble for the initial time step. If not, the implementation is erroneous. To assemble the cell set of the bubble, execute the `testFieldCellSet` application in the `rising-bubble-2D` example case directory:

```
?> testFieldCellSet -field alpha1
```

The resulting cell set is stored in the `0` directory and to visualize it using the `paraView` application, it must be converted to the VTK format using `foamToVTK`. Before this can be done, the cell set must be copied from the `0` directory to `constant/polyMesh/sets`:

```
?> mkdir -p constant/polyMesh/sets
?> cp 0/fieldCellSet !$
```

Once this is done, the cell set itself can be converted:

```
?> foamToVTK -cellSet fieldCellSet
```

Which will generate the VTK directory within the `rising-bubble-2D` example case directory. In this directory, the field data related to the cells agglomerated by the `fieldCellSet` class will be stored, so we can visualize them. Figure 12.1 shows the `alpha1` field of the rising bubble cell set for the initial time step. The computation and visualization of the field based cell set could have easily be done in the `paraView` application, but the point is to have the understanding of function objects in C++, and how they may be easily used when programming OpenFOAM code. Function objects in C++ are very much like extended functions from the

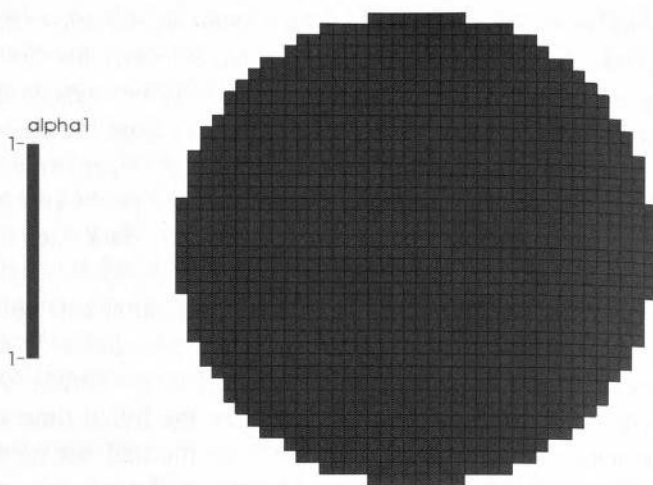


Figure 12.1: Field `alpha1` of the field based cell set comprising the rising bubble in the initial time step.

perspective of the language expressions, whereas the function objects in OpenFOAM have similar functionality to one shown in this example but very different design and capabilities.

12.1.2 Function Objects in OpenFOAM

Function objects in OpenFOAM have a different class interface than standard function objects encountered in C++ have. Instead of relying on overloading `operator()`, they adhere to a class interface defined by the `functionObject` abstract class, as shown in listing 100. Which in turn results in a class hierarchy, structurally similar to the hierarchy used for boundary conditions (chapter 10) and transport models (chapter 11). There is an abstract base class (`functionObject`) which prescribes the interface of all function objects in OpenFOAM. Compared to the standard C++ function objects, this extended class interface enables OpenFOAM function objects to be executed at various events of a simulation.

The execution of the `functionObject` member functions is related to either changes of the simulation time or changes of the mesh, as shown in listing 100. Simulations in OpenFOAM are controlled by the `Time` class and the majority of the member functions of `functionObject` have

Listing 100 Class interface of the function object abstract base class

```

// Member Functions

// - Name
virtual const word& name() const;

// - Called at the start of the time-loop
virtual bool start() = 0;

// - Called at each ++ or += of the time-loop.
// forceWrite overrides the outputControl behaviour.
virtual bool execute(const bool forceWrite) = 0;

// - Called when Time::run() determines that
// the time-loop exits.
// By default it simply calls execute().
virtual bool end();

// - Called when time was set at the end of
// the Time::operator++
virtual bool timeSet();

// - Read and set the function object rf
// its data have changed.
virtual bool read(const dictionary&) = 0;

// - Update for changes of mesh
virtual void updateMesh(const mapPolyMesh& mpm) = 0;

// - Update for changes of mesh
virtual void movePoints(const polyMesh& mesh) = 0;

```

something to do with the change in the simulation time. Thus the `Time` class is made responsible for invocation of `functionObject` member functions, depending on the event. Two member functions that relate to mesh motion (`movePoints`) and mapping of fields (`updateMesh`) are called from within the mesh class `polyMesh`, using the constant access to simulation time.

As the result of those requirements, the `Time` class composites all function objects loaded for each particular simulation in a list of function objects (`functionObjectList`) as shown in the diagram in figure 12.2. The `functionObjectList` implements the interface of `functionObject` and delegates the member function invocation to the composited function objects. As the simulation starts and the `runTime` object of the `Time` class gets initialized, the simulation control dictionary `controlDict` is

read. The constructor of `functionObjectList` reads the `controlDict` and parses the entries in the `functions` sub-dictionary. Each entry in the `functions` sub-dictionary defines parameters of a single function object, which are then passed to the function object selector. The selector (`functionObject::New`) implements the "Factory Pattern" known from OOP and uses the dictionary parameter type to initialize a concrete model of the `functionObject` abstract class during runtime. Finally, the selected function object gets appended to the list of function objects. This mechanism also relies on the RTS mechanism in OpenFOAM, allowing the user to select and instantiate different function objects for different simulation cases.

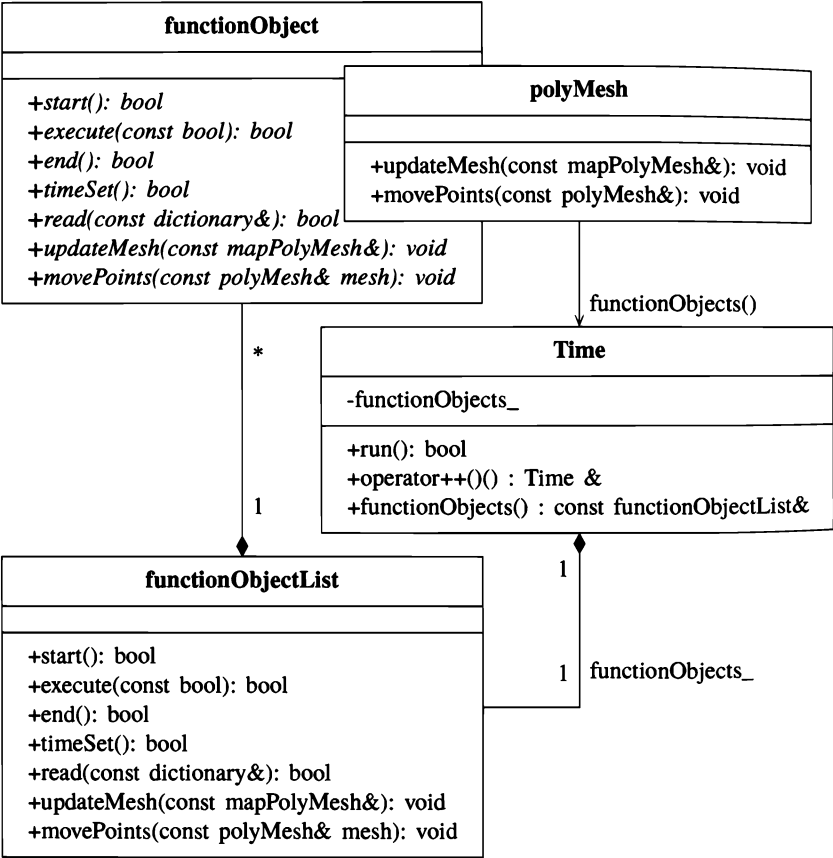


Figure 12.2: Composition of function objects in the `Time` class.

Since the function objects are initialized during runtime and they rely



on dynamic polymorphism (virtual functions), the member functions that implement the main functionality have an overhead when resolving which virtual member function is to be called (dynamic dispatch). However, with the operations of function objects taking orders of magnitude more computational time than dynamic dispatch, this property of function objects can be neglected with good faith.

The function objects in OpenFOAM are also objects that perform function-like operations during the simulation, so their name can be justified by that property. The class declaration of `functionObject` serves as another example to describe further differences:

```
// Private Member Functions

//- Disallow default bitwise copy construct
functionObject(const functionObject&);

//- Disallow default bitwise assignment
void operator=(const functionObject&);
```

Both, the assignment and copy construction is prohibited for function objects in OpenFOAM, making it impossible to pass them as value arguments. C++ function objects on the other hand can be handled like this. This has no impact for their use, since they are not meant to be used in that way: The central point of use of function objects in OpenFOAM is the private attribute `functionObjectList` in the `Time` class. There is no need to instantiate them manually.

Having described the design of `functionObject` and the interaction with `Time`, the remaining point is what components are necessary for programming new function objects. Basically any class that inherits from `functionObject`, implements its interface and whose type entry as well as necessary parameters are listed within the `controlDict` simulation control directory will be interpreted as a function object in OpenFOAM. This entire process is triggered by `Time` and hence happens automatically. However, before developing a new function object, it is advisable to check if the desired functionality is already distributed with the official OpenFOAM release, or in a community contribution. Both are described briefly in the following section.

12.2 Using OpenFOAM Function Objects

Before starting with the development of a new function object, it is recommended to check if the desired functionality is already existent. Besides the function objects in the official release, there are various community driven developments, that provide a large variety of different functionalities. When it comes to function objects, the most significant community contribution are the function object libraries of the `swak4foam` project, originally developed by Bernhard Gschaider.

The falling droplet example is reused for the explanation of the function objects in OpenFOAM official release and those of the `swak4foam` project.

12.2.1 Function Objects in the Official Release

The categorization of function objects in the official release of OpenFOAM is described in detail in the online Doxygen documentation under the Modules category, and is hence omitted. The source code of the majority of function objects in the official release is located in `$FOAM_SRC/postProcessing/functionObjects`.

Function objects contained in the official release can be used by providing the adequate entry in the `system/controlDict` of the simulation case. Parameters required by the function object and the library which implements the function object must be specified in the `controlDict`.

In order to illustrate the usage of a rather simple function object, the `CourantNo` function object is selected. It computes the Courant number for every cell in the mesh and stores it as a `volScalarField`, that is then saved for further analysis. To use the `courantNumber` function object, the dynamically loaded library and the `functions` entries in the `system/controlDict` file of the `falling-droplet-2D` simulation case need to be defined as shown in listing 101. For the `courantNo` function object, as well as for other function objects in the official OpenFOAM release, the output control operations are implemented independently of the simulation output control in the `controlDict` file. Because of this, the additional entry `outputControl` is necessary to prescribe the output



Listing 101 Definition of the CourantNo function object

```

libs ("libutilityFunctionObjects.so");

functions
{
    courantNo
    {
        type CourantNo;
        phiName phi;
        rhoName rho;
        outputControl outputTime;
    }
}

```

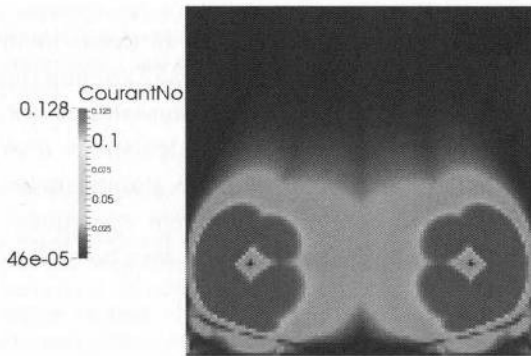


Figure 12.3: Courant number distribution for the falling droplet simulation case.

control of the function object. For the sake of this example, the set in this case to `outputTime`. This setting results in the Courant number field being written to the case, whenever the other fields are written anyway. Figure 12.3 shows the resulting Courant number distribution at 1.5 seconds of simulation time for the simulation run with the `interFoam` solver. Since the mesh density is uniform for this simulation case, it is obvious that there is an intense transfer of energy from the falling droplet to the surrounding air, that in turn gets accelerated in two separate vortices.

12.2.2 Function Objects in swak4foam

The swak4foam project is mentioned throughout the book, since it offers many interesting functionalities to the OpenFOAM user. So is the case with function objects, which are implemented in `swakFunctionObjects`, `swakFunctionObjects` and `simpleSwakFunctionObjects` libraries. All three libraries together number up to more than 50 function objects, which makes them the largest function object community contribution to OpenFOAM and is thus well worth investigating and using. Since the libraries hold many useful function object implementations, this section is prepared to make the users of OpenFOAM aware of this powerful library, by providing a single example of an interesting function object.

The installation of swak4foam is described in detail in the OpenFOAM wiki¹ page, so it is skipped here. The `swakExpressionFunctionOb-`
`ject` is chosen among many useful function objects for this example, because of its versatility: it supports field calculations involving field expressions defined as strings. To put it more simply: it is a string based field calculator that allows computing different operations with fields involved in the simulation, agglomerating the results, and storing them for further analysis.

An expression involving fields could be written as:

```
(U & U) * 0.5 + 9.81 * pos().z + p / rho
```

This expression represents the Bernoulli equation. Expressions such as the Bernoulli equation are analyzed using an *expression parser* that separates the field operators (`&`, `+`, `/`, `^`, `-`, `...`) in OpenFOAM, from the field names (`U`, `p`, `rho`, and so forth). Once the operator and field names are separated, the function object performs a mapping between the strings that define the operator and field names, and the actual operators and fields in OpenFOAM, which allows for the expression to be evaluated.

¹<http://openfoamwiki.net/index.php/Contrib/swak4Foam>



Computing Droplet Kinetic and Potential Energy

The `swakExpressionFunctionObject` is used to evaluate the kinetic and potential energy of a falling droplet.

In order to compute the kinetic and potential energy of the droplet phase, the following code snippet needs to be added to the `system/controlDict` of the simulation case:

```
libs ("libsimpleSwakFunctionObjects.so");

functions
{
    dropletKineticEnergy
    {
        type swakExpression;
        expression "alpha1*rho*vol()*(U & U)*0.5";
        accumulations (sum);
        valueType internalField;
        outputControlMode outputTime;
    }

    dropletPotentialEnergy
    {
        type swakExpression;
        expression "alpha1*rho*vol()*9.81*pos().y";
        accumulations (sum);
        valueType internalField;
        outputControlMode outputTime;
    }
}
```

It shows that the `swakExpressionFunctionObject` is compiled into the `simpleSwakFunctionObjects` library which needs to be loaded at runtime. For this simulation case two expressions are used to compute the kinetic and the potential energy of the liquid phase, that represents the droplet. The results of the computation can be processed into a single value, by specifying the `accumulations` entry in the function object's sub-dictionary. To compute the total energy of phase 1, the `sum` operation is specified as an accumulation. Alternative operations can be selected as well:

- `average`
- `max`
- `min`

- `weightedAverage`

Any computation is performed on the `internalField` of the field it should operate on. Subsets of the mesh can be selected by the following topological operators:

- `cellSet`
- `cellZone`
- `faceSet`
- `faceZone`
- `patch`
- `set`
- `surface`

The output frequency of the function object is controlled by the `output-ControlMode` `outputTime` keyword and it is separated from the output frequency as well as the control of the simulation case. Alternative output modes for the function objects are `timestep` and `deltaT`. When the output frequency of the `swakExpressionFunctionObject` is set to `timestep`, an additional parameter is required:

```
outputInterval N;
```

In this case, `N` defines the number of time steps after which the output is generated. If the `deltaT` output control mode is selected, the `output-DeltaT` entry is required and represents the value of the output time step in seconds. A value of 0.5 instructs the function object to write the data every 0.5 seconds of simulation time:

```
outputDeltaT x;
```

To execute the `swakExpressionFunctionObject` and compute the kinetic and potential energy of the falling droplet, the mesh needs to be created for the `falling-droplet-2D` simulation case and the volume fraction field must be initialized:

```
?> blockMesh
```

and the initialization of the volume fraction field for the droplet is neces-



sary:

```
?> setFields
```

After these steps, the `interFoam` solver can be executed:

```
?> interFoam
```

After the simulation is completed, a directory named `postProcessing` appears in the case directory, containing two subdirectories:

- `swakExpression_dropletKineticEnergy`
- `swakExpression_dropletPotentialEnergy`

They contain the `dropletKineticEnergy` and the `dropletPotentialEnergy` data files, respectively.

Figure 12.4 shows the air acceleration caused by the motion of the falling droplet. The potential energy of the droplet, that is initially placed above the bottom wall is converted to kinetic energy of the droplet, but also the surrounding air. Another part of the potential energy gets dissipated due to viscous effects of both phases. The temporal development of both kinetic and potential energy for the `falling-droplet-2D` simulation case are shown in figure 12.5.

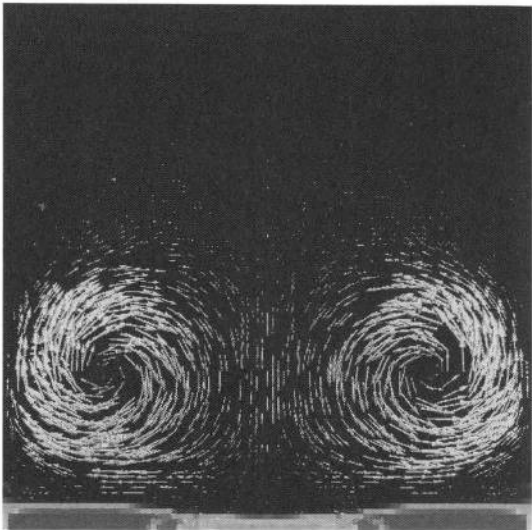


Figure 12.4: Air accelerated by the movement of the falling droplet.

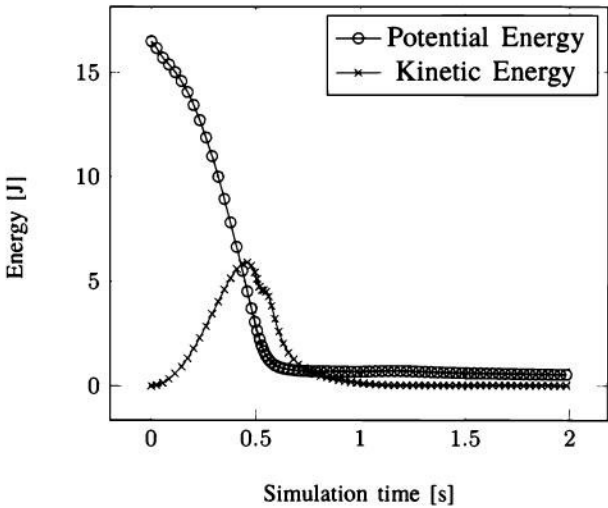


Figure 12.5: Kinetic and potential energy of the droplet (phase 1).



12.3 Implementation of a Custom Function Object

Since the implementation of a new function object in its basis consists of implementing the interface of the `functionObject` abstract class, a shell script is available in the code repository, that produces the necessary file structure for a new function object library. The script is named `newFunctionObject` and it is distributed along with the code examples in the `src/scripts` sub-directory of the example code repository. This includes a directory structure, empty template files and the essential make directives.

12.3.1 Function Object Generator

Before using the function object generator script in the shell, make sure that the OpenFOAM environmental variables are set. To create a new function object library with the `newFunctionObject` script, the directory that will hold the library files needs to be created manually:

```
?> mkdir exampleFunctionObjectLibrary
?> cd !$
```

After the directory is created successfully, a new function object can be added, using the script:

```
?> newFunctionObject -d myFunctionObject myFunctionObject
```

The `-d` option of the `newFunctionObject` generator script is responsible for creating a sub-directory, where the `.H` and the `.C` files of the function objects will be stored for better code organization. Listing the directory contents shows the new file structure created by the `newFunctionObject` script:

```
?> ls *
Make:
files options

myFunctionObject:
myFunctionObject.C myFunctionObject.H
```

By default, the new function object `myFunctionObject` inherits from `functionObject`, which can be easily changed in a text editor. It can be placed somewhere different in the function object class hierarchy. Viewing the file `Make/files`:

```
?> cat Make/files
myFunctionObject/myFunctionObject.C
LIB = $(FOAM_USER_LIBBIN)/libexampleFunctionObjectLibrary
```

It shows that the compiled library file will be stored in the user library directory defined by the OpenFOAM platform environment, which makes the library available automatically. The library will be named after the directory that stores the library files, following the standard practice in OpenFOAM (`libexampleFunctionObjectLibrary`). To compile the library, execute `wmakelibso`. Once the library is compiled successfully, the function object `myFunctionObject` can be used with any solver and any simulation case, provided that the library is added to the list of dynamically loaded libraries in the `controlDict` file.

To see what the new function object actually does, it can be tested with any case and solver. For this purpose, the `cavity` tutorial case and the `icoFoam` solver are selected. As usual, the tutorial case is copied to the user simulation run directory and the mesh is generated using the `blockMesh` application:

```
?> run
?> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity .
?> cd cavity
?> blockMesh
```

In order to use the new function object, the following entries need to be added to the `system/controlDict` file:

```
libs ("libexampleFunctionObjectLibrary.so");

functions
{
    myFunctions
    {
        type myFunctionObject;
    }
}
```



On execution of the `icoFoam` solver, a *a fatal execution error* should be raised like the following:

```
--> FOAM FATAL ERROR:
Not implemented
```

```
From function myFunctionObject::start()
in file myFunctionObject/myFunctionObject.C at line 85.
```

This is expected since the generated function object class `myFunctionObject` implementation contains only error inducing stubs for the member functions that are yet to be implemented (see `myFunctionObject.C`), as shown in listing 102.

Listing 102 Bare implementation of `myFunctionObject`

```
bool myFunctionObject::read(const dictionary& dict)
{
    notImplemented("myFunctionObject::read(const dictionary&)");
    return execute(false);
}

bool myFunctionObject::execute(const bool forceWrite)
{
    notImplemented("myFunctionObject::execute(const bool)");
    return true;
}

bool myFunctionObject::start()
{
    notImplemented("myFunctionObject::start()");
    return true;
}

bool myFunctionObject::end()
{
    notImplemented("myFunctionObject::end()");
    return execute(false);
}
```

The member functions are to be implemented by the programmer, depending on the desired functionalities. The fatal execution error shows that the library is successfully loaded and that the `myFunctionObject` class constructor is successfully added to the RTS constructor table of the `functionObject` abstract class.

12.3.2 Implementing the Function Object

The goal of this example is not to implement a function object useful for CFD calculations within OpenFOAM: both OpenFOAM and the swak4foam project already contain a large number of useful function objects. On the contrary, the goal of this example is to describe what parts of the `functionObject` class interface are important to consider when implementing a custom function object.

A function object runtime operations can be kept separate from the calculation the function object performs. In case the function object calculation is implemented into the function object itself, the calculation can only be used by the function object. Encapsulating the calculation in a class and refactoring it out of the function object allows other clients to use it, e.g. post-processing applications. Also, de-coupling the calculation from the function object runtime processing capability leads to a more clean implementation with separated concerns: the class that calculates something is separate from the function object class that deals with the runtime execution. When a new function object is programmed, considering re-using existing classes in OpenFOAM and coupling them with `functionObject` via either composition or public inheritance, depending on the situation might also save development time. In the example shown in this section, the function object class implements the calculation, and inherits the runtime operation from `functionObject`.

Member functions of the `functionObject` abstract class that need to be implemented when the new function object is programmed are:

- `start` : executed at the beginning of the time loop,
- `execute` : executed when the time is increased,
- `end` : executed when the time-loop ends (time has reached the value of `endTime`).

In case when the calculation is not different at the beginning and the end of the time loop, member functions `start` and `end` call the member function `execute`. The function object described in this section keeps track of the cells in the mesh that got wetted with a specific phase in a multiphase simulation. In each time step of the simulation, it checks the value of the volume fraction field in each cell, and marks the cells with



the value greater than zero by storing the value 1 as the cell value of the marker field. Visualizing this field shows every cell of the mesh that got filled with a specific phase during the course of the simulation.

The code for the function object example is available in the example code repository, and viewing the code in a text editor may help with understanding of the example. The names of the directories in this example are named the same way as the directories of the code example repository. Before using the `newFunctionObject` generator in a shell console, make sure that the OpenFOAM environmental variables have been properly set. That the example code repository also needs to be configured by sourcing the `primer-examples/etc/bashrc` configuration script. To start programming the library, create a directory named `primerFunctionObjects` for the new function object library. The new function object library code can at this point be created using the `newFunctionObject` generator:

```
?> newFunctionObject \  
    primerFunctionObjects/wettedCellsFunctionObject \  
    wettedCellsFunctionObject
```

To test if the generated code compiles, execute `wmake`:

```
?> wmake
```

If the compilation resulted without errors, the generated dummy code is ready to be expanded with the actual implementation.

For marking the wetted cells, the `wettedCellsFunctionObject` requires a `volScalarField`, a user prescribed tolerance, and a percent of the domain which has been wetted so far. The following private attributes need to be present in the class declaration in `wettedCellsFunctionObject.H`, as shown in listing 103. Once the private attributes are declared, they need to be initialized by the class constructor in listing 104. In order to be able to select the `wettedCellsFunctionObject` during runtime, the `TypeName` macro needs to be modified as well:

```
// - Runtime type information  
TypeName("wettedCells");
```

Listing 103 Private attributes of `wettedCellsFunctionObject`

```
// Private data

//- Time
const Time& time_;

//- Mesh reference;
const fvMesh& mesh_;

//- Volume fraction field.
const volScalarField& alpha1_;

//- Wetted cells field.
autoPtr<volScalarField> wettedCells_;

//- Wetted cell tolerance.
scalar wettedTolerance_;

//- Wetted domain percent.
scalar wettedDomainPercent_;
```

Having the runtime type information declared properly, the function object is made selectable by providing an entry in the `system/controlDict` dictionary. To test the integration of the function object with OpenFOAM, the falling-droplet-2D test case of the example case repository is used. To test the function object, the following sub-dictionary is to be added to the functions dictionary within `controlDict`:

```
wettedCellsTest
{
    type wettedCells;
    volFractionField alpha1;
}
```

and the library needs to be loaded during runtime, which is done by adding the following to the list of loaded libraries, also in the `controlDict` file:

```
libs (
    "libsimpleSwakFunctionObjects.so"
    "libutilityFunctionObjects.so"
    "libofBookFunctionObjects.so"
);
```

In fact, to test the `wettedCellsFunctionObject` only the `"libofBookFunctionObjects.so"` is required, the other libraries are coming from



Listing 104 Initialization of data members of `wettedCellsFunctionObject`

```

functionObject(name),
time_(time),
mesh_(time.lookupObject<fvMesh>(polyMesh::defaultRegion)),
alpha1_
(
    mesh_.lookupObject<volScalarField>
    (
        dict.lookup("volFractionField")
    )
),
wettedCells_
(
    new volScalarField
    (
        IOobject
        (
            "wettedCells",
            time.timeName(),
            time,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimensionedScalar
        (
            "zero",
            dimless,
            0.0
        )
    )
),
wettedTolerance_(readScalar(dict.lookup("wettedTolerance"))),
wettedDomainPercent_(0)

```

the previous examples described within this chapter. To execute testing, start the `interFoam` solver, which should end in a `FatalError` similar to this one:

```

--> FOAM FATAL ERROR:
Not implemented

```

```

From function wettedCellsFunctionObject::start()

```

This happened because the member functions of `wettedFunctionObject` need yet to be implemented, with `start()` being the first one called during the simulation:



```
bool wettedCellsFunctionObject::start()
{
    notImplemented("wettedCellsFunctionObject::start()");
    return true;
}
```

Before continuing with the implementation, the `timeSet` member function is to be removed from the function object, since it is not needed. For this example function object the calculation does not differ at the beginning of the simulation and at the end, so both the `start` and `end` member functions re-use the `execute` member function:

```
bool wettedCellsFunctionObject::start()
{
    execute(false);
    return true;
}

bool wettedCellsFunctionObject::end()
{
    return execute(false);
}
```

Modifications performed on the mesh, involving motion or topological changes, also do not modify the semantics of the calculation, so the `updateMesh` and `movePoints` also re-use the `execute` member function:

```
void wettedCellsFunctionObject::updateMesh(const mapPolyMesh& map)
{
    execute(false);
}

void wettedCellsFunctionObject::movePoints(const polyMesh& mesh)
{
    execute(false);
}
```

The `execute` member function is where the calculation and reporting of the current status of the function object are performed, and it is implemented in the following way:

```
bool wettedCellsFunctionObject::execute(const bool forceWrite)
{
    calcWettedCells();
    calcWettedDomainPercent();
    report();
}
```



```

    return true;
}

```

The calculations and the report have been separated and refactored into member functions, which makes it easier to modify the semantics of the execution by inheriting from the `wettedCellsFunctionObject`. As their names state, `calcWettedCells`, `calcWettedDomainPercent` and `report` member functions calculate the wetted cells, calculate the percentage of the wetted cells in the flow domain, and report on the function object state. They have all been declared as virtual member function, to support extensions using inheritance without modifying the `wettedCellsFunctionObject`. Note that, although the `wettedCells` function object is simple and self-explanatory, we still aimed at applying object oriented designs and principles that allow future extensions without modification of the existing class. The actual calculation that marks cells as wetted cells is straightforward and shown in listing 105. In listing 105,

Listing 105 Algorithm to mark cells as wetted

```

void wettedCellsFunctionObject::calcWettedCells()
{
    volScalarField& wettedCells_ = wettedCellsPtr_();

    forAll (alpha1_, I)
    {
        if (isWetted(alpha1_[I]))
        {
            wettedCells_[I] = 1;
        }
    }
}

```

`isWetted` denotes the member function that checks the volume fraction value against the user prescribed tolerance:

```

bool isWetted(scalar s)
{
    return (s > wettedTolerance_);
}

```

The calculation of the total percentage of the domain that has been wetted during the course of the simulation is implemented as in listing 106.

The value of the `volScalarField` named "wettedCells" will be set to 1 for cells that are wetted. The function simply checks which cells are

Listing 106 Calculation of relative surface wetting

```
void wettedCellsFunctionObject::calcWettedDomainPercent()
{
    scalar wettedCellsSum = 0;

    const volScalarField& wettedCells = wettedCellsPtr_();

    forAll (wettedCells, I)
    {
        if (wettedCells[I] == 1)
        {
            wettedCellsSum += 1;
        }
    }

    wettedDomainPercent_ = wettedCellsSum / alpha1_.size() * 100;
}
```

wetted and computes a ratio of the wetted cells and the total number of cells in the domain. This value is then stored as the `wettedDomainPercent_` private attribute and is reported on by the `report` member function:

```
void wettedCellsFunctionObject::report()
{
    Info << "Wetted " << wettedDomainPercent_
        << " % of the domain." << endl;
}
```

For testing purposes, the mesh resolution of the falling droplet test case has been reduced. When the case is run with the original higher mesh resolution, the results are expected to behave in the same way, qualitatively.

The "wettedCells" field stored by the function object results is shown for the falling droplet simulation case on figure 12.6 together with the α_1 volume fraction field for the same time step. Figure 12.7 shows the plot of the wetted domain percentage against the simulation time.



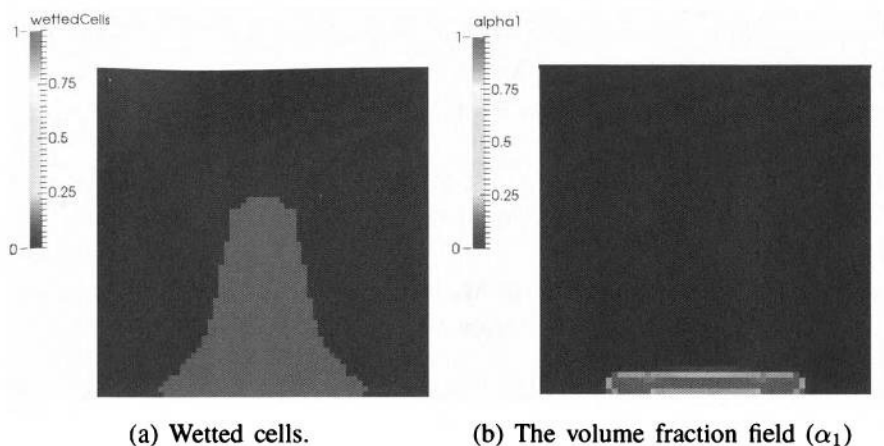


Figure 12.6: Wetted cells and the volume fraction field for the falling droplet test case.

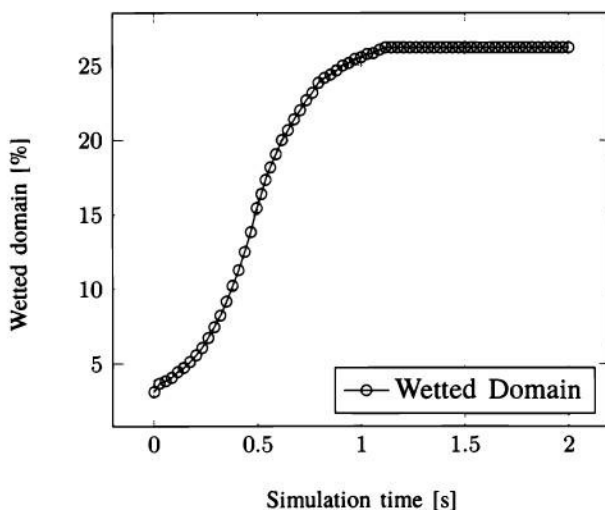


Figure 12.7: Percentage of the wetted domain.

EXERCISE

An interesting exercise would be to implement the wetted domain percentage calculation in parallel. What happens when the function object is run on a parallel case? Hint: take a look at the `Pstream` class.

Further reading

- Alexandrescu, Andrei (2001). *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Josuttis, Nicolai M. (1999). *The C++ standard library: a tutorial and reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Vandevoorde, David and Nicolai M. Josuttis (2002). *C++ Templates: The Complete Guide*. 1st ed. Addison-Wesley Professional.



13

Dynamic Mesh Operations in OpenFOAM

Generally there are two main dynamic mesh handling capabilities in OpenFOAM: *mesh motion* and *topological changes* (H. Jasak and Rusche 2009, H. Jasak and Tukovic 2006). Mesh motion is a dynamic mesh operation that solely involves displacement of the mesh points without altering the topological information of the mesh. The topological information describes how the points, edges, faces and cells of the mesh are built, as well as the way those mesh elements relate to each other. At first glance, displacing mesh points looks like a rather trivial task, but depending on what motion is desired, the operations can be more complex than expected. As the geometry of mesh faces and cells are based upon the mesh points, they deform as a result of the motion of the points. After the mesh has been deformed, the fields whose values still relate to the initial mesh need to be mapped to the new mesh. This is necessary since the field values in FVM represent averages with respect to the cell volume, or face area. Both the cell volume and the area of the face may change as a result of mesh motion.

Changing the topology of the mesh often includes changing the number of points and consequentially changing the mesh elements (cells, faces,

edges) as well as the connectivity between the mesh elements. Considering this, the operations involving topological mesh changes are rather complex, when compared to mesh motion, as they involve more complex algorithms and data structures.

There are two main problem categories, for which topological changes are necessary to obtain a fast and accurate solution: moving mesh boundaries and presence of large gradients in the mesh.

When an object is moving significantly inside the domain and relative motion exists between mesh points, the cells are likely to get either too distorted or compressed. An example for this is a piston moving in a cylinder: layers of cells are added or removed, parts of the mesh can be disconnected from each other and be reconnected later on.

The second problem category requires the simulation to have a higher accuracy in those domain regions that are unknown a priori, e.g. at a point when the mesh is generated. For simulations that involve shocks in the simulation domain where the position of the shock is a part of the solution process, topological changes are applied based on e.g. pressure gradient in order to achieve local static refinement in the region where the shock appears. A different example is an interface between two immiscible liquid phases in a two-phase simulation: physical properties change abruptly in values across the interface, but the interface position is known at the start of the simulation. However, in order to obtain a more accurate solution, mesh is locally and dynamically refined in the near vicinity of the fluid interface, and the refinement follows the interface as it moves across the computational domain.

Keeping the dynamic mesh operations on a higher level of abstraction by encapsulating them into classes of a class hierarchy, allows the OpenFOAM user to detach the dynamic mesh operations from a flow solver. Dynamic mesh classes can also be combined using object oriented design principles, in order to extend the flow solver with multiple dynamic mesh operations to increase both accuracy and flexibility of a numerical simulation.

An overview of the existing types of dynamic meshes is provided in this chapter, and a set of chosen dynamic mesh engines available in Open-



FOAM are described in more detail. Extensive details about the design of the base classes' design as well as some selected dynamic meshes can be found in section 13.1. Of more interest for the reader interested in using dynamic mesh handling is the section 13.2, that presents some usage examples. Extending the existing dynamic mesh handling available in OpenFOAM by combining solid body motion and hexahedral mesh refinement is described in section 13.3.

13.1 Software Design

The design of the dynamic mesh classes varies depending on the functionality of each particular dynamic mesh. In this chapter, a brief overview of the functionality and the design of dynamic meshes in OpenFOAM is provided, involving mesh motion and topological changes of the mesh.

13.1.1 Mesh Motion

There are two main types of mesh motion operations implemented in OpenFOAM: *solid body mesh motion* and *mesh deformation*.

As the name suggests, the solid body motion involves displacement of mesh points, which is performed in a way that the points retain their relative position. Solid body mesh motion involves translation and rotation of the mesh, or a combination of both. It can either be prescribed, or computed as a solution of ordinary differential equations that model the dynamics of the solid body motion.

Mesh deformation on the other hand relies on distributing the displacement from the mesh boundary to the inner parts of the flow domain. The displacement distribution can be performed using various methods, involving algebraic interpolation or solving of transport equations for the point displacement or point velocity. Both solely determine how a certain motion will be evaluated at the mesh points, but not how the motion itself is defined: using either point velocity or displacement.

Solid Body Mesh Motion

The solid body mesh motion is defined as a motion of the body, applied to the mesh points, where no relative displacement between any two points occurs. Such a motion is sketched in figure 13.1, with the filled body being the solid body. Before going into detail of how a certain motion is defined, the definition of the motion itself and its implementation is described.

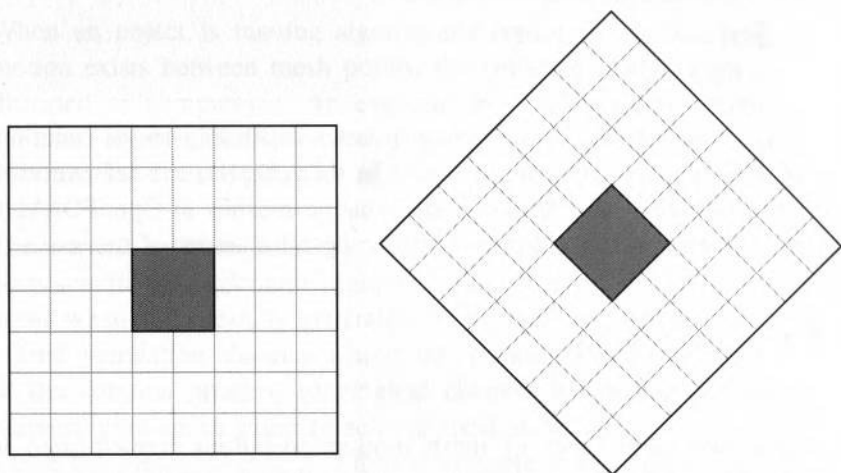


Figure 13.1: The filled body is subjected to a solid body motion, so the mesh points do not move relatively to each other.

In OpenFOAM, the solid body motion is defined by a variety of classes, all derived from their common base: `solidBodyMotionFunction`. The motion function returns a `septernion` which describes the motion of the body. This motion is a combination of a vector for the translation and a quaternion for the rotation. A `septernion` can be used easily to move each point using the same vector transformation operation, involving a set of multiplications. More information about quaternions can be found in the book by Goldman 2010.

Classes derived from the abstract base class `solidBodyMotionFunction` define what type of motion is present. Ranging from simple ones, such as `linearMotion` and `rotatingMotion` over more sophisticated functions, like `tabulatedMotion`, to complex and rather specific motions (SDA

for seakeeping investigations of ships). As can be seen from the UML diagram in figure 13.11, the `solidBodyMotionFunction` defines the virtual method `transformation` that in turn is implemented by each derived class. It calculates the septonion used for to transform the mesh points between the initial position and the position at the current time and represents both the translation as well as the rotation.

The transformation function itself is encapsulated in a class in order to make it runtime selectable and easily re-usable. None of the classes derived from `solidBodyMotionFunction` do change the location of the mesh points, they just provide the transformation.

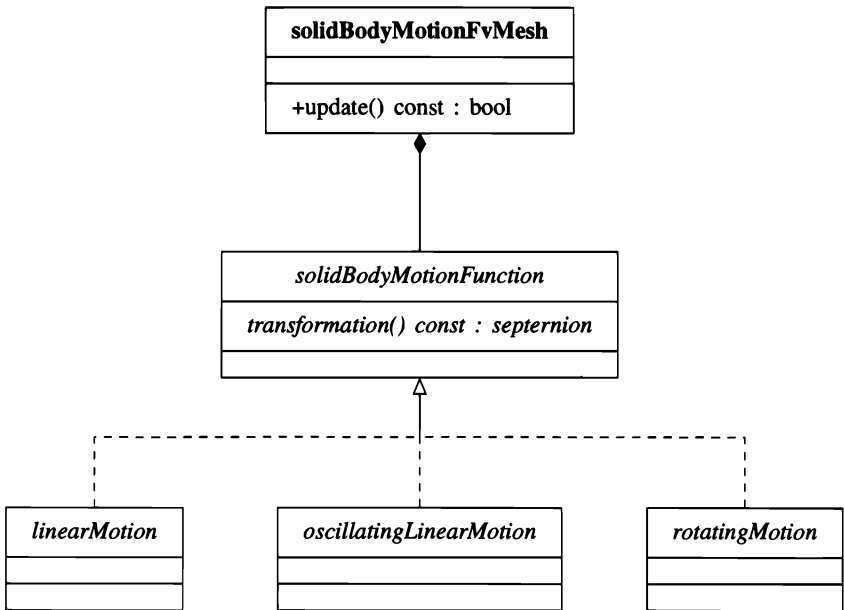


Figure 13.2: Class diagram for the `solidBodyMotionFvMesh` and the `solidBodyMotionFunction`.

The dynamic mesh class on the other hand, that performs the actual solid body mesh motion is named `solidBodyMotionFvMesh` and inherits from `dynamicFvMesh`. The relationship between `solidBodyMotionFvMesh` and `solidBodyMotionFunction` is shown on the class diagram in figure 13.2.

The transformation function is implemented using the Strategy Pattern

(Gamma, Helm, Johnson, and Vlissides 1995) for the `solidBodyMotionFvMesh` class. As a consequence, the solid body motion function

TIP

Strategy pattern:

Encapsulating various algorithms in a class hierarchy, composing them in the client (user) class, and making each of them selectable during runtime.

can be re-used by other parts of the library as a standalone entity, without any problems. In order to compute the solid body motion, the relationship between a class and the solid body mesh motion must not be known. Additionally, the `solidBodyMotionFunction` is designed using the Strategy pattern and allows for an easy addition of other functions. Simply inheriting from `solidBodyMotionFunction`, the new functions are made runtime selectable.

If, on the other hand, the `solidBodyMotionFunction` would be implemented as a Template Method pattern in the `solidBodyMotionFvMesh` class, adding another mesh motion function would result in having one more dynamic mesh class. Example of the Template Method pattern application is the `dynamicFvMesh` class and the derived dynamic mesh classes, implementing the update algorithm.

TIP

Template Method pattern:

A virtual function is used to implement the algorithm and the (base) class may provide a default algorithm implementation. As a result, the derived classes implement the variety of alternative algorithms, resulting in a per-algorithm branching in the hierarchy.

The design of the `solidBodyMotionFunction` class hierarchy is a good example for the SRP design principle: even though the motion functions are fairly simple, they are encapsulated as individual classes having a single simple responsibility. The motion function is concerned with computing the transformation and nothing else. Additionally to that, the motion functions could generally be combined: e.g. an oscillating linear motion



with constant rotation. In that case, the Template Method pattern breaks down, as it relies on using multiple inheritance with the dynamic mesh class. The problems involved with multiple inheritance and dynamic mesh classes are addressed in section 13.3.2.

The `solidBodyMotion` dynamic mesh transforms either all points of the mesh, or a specific part (i.e. a `cellZone`), using the same septernion defined by the user selected `solidBodyMotionFunction`. If no `cellZone` is specified, all points in the mesh are moved using the same septernion, hence a uniform motion is achieved. If a `cellZone` is specified in the dictionary, only the points of the cells in the `cellZone` are moved accordingly. This comes in handy, when a body is rotating in a rotor-stator configuration, using sliding interfaces (AMI/GGI). In addition to the `solidBodyMotionFvMesh`, there is the `multiSolidBodyMotionFvMesh`, which basically does the same as `solidBodyMotionFvMesh`, but allows for multiple motions to be either superimposed or act upon different `cellZones`. The latter is very important for applications where two rotors rotate, using different `cellZones` and AMI/GGI interfaces. The motion itself is applied to all relevant mesh points in a direct fashion, using `fvMesh::movePoints`, so no extra equations are solved, which makes this a fairly fast approach.

TIP

Use `cellZone` based selection of mesh points for a rotor-stator mesh motion configuration that involves a sliding interface.

Mesh Deformation

Mesh deformation, or mesh morphing, is performed by applying a different displacement to each mesh point. Mesh deformation will introduce relative motion between mesh points, which will in turn distort mesh cells and modify the mesh in an inhomogeneous way. This usually leads to high distortions, especially for larger motions, which often reduces the quality of the mesh.

In order to maintain the mesh quality, the mesh deformation needs to be used carefully: often only a small sub-region of the mesh is deformed strongly, while the deformation for the rest of the mesh is kept as low

TIP

The way the quality of the mesh is determined depends on the numerical method. For the FVM, two most important discretization errors that are mesh related are the non-orthogonality and the skewness error (Jasak 1996, Juretić 2004).

as possible. This kind of approach results in mesh motion in the region where it is necessary, which is often in the vicinity of the mesh boundary, or a part of the mesh boundary. Additionally, the mesh deformation can be regarded as an optimization problem, where the quality of the mesh represents a domain-global optimized scalar function.

The mesh boundaries define the motion and are therefore the locations with the largest motion present. Having the strongest displacement being applied on the specific part of the mesh boundary, the remaining question is how to propagate the displacement to the rest of the mesh, while keeping the distortion minimal in the regions of no interest for mesh motion.

Two most prominent approaches are available for this purpose: algebraic displacement interpolation and solution of a diffusion (Laplace) transport equation for the displacement. In OpenFOAM, both the algebraic displacement interpolation as well as the mesh motion solver approach are available.

The description of the interpolation for the mesh point displacement based on Radial Basis Function (RBF) is provided by Bos, Matijasevic, Terze, van Oudheusden, and Bijl 2008, Bos 2009, and Bos, Oudheusden, and Bijl 2013. The choice of the interpolation may significantly improve the accuracy of the solution. In case of the RBF interpolation, the mesh deformation results in a better overall quality of the mesh, as well as a better quality of the boundary layer mesh.

The mesh motion that relies on a diffusion of the boundary mesh displacement to the internal parts of the solution domain can be modeled as

$$\nabla \cdot (\gamma \nabla \mathbf{d}) = 0, \quad (13.1)$$



WARNING

The RBF interpolation and the mesh quality based mesh motion are topics that require more attention and are therefore placed outside of the scope of this book.

where γ is the displacement diffusion coefficient, and \mathbf{d} is the point displacement field. The rate of diffusion of the displacement from the boundaries to the internal mesh regions is defined with the spatially varying diffusion coefficient γ given as a function of the distance between the point \mathbf{x} and the mesh boundary, i.e.

$$\gamma = \gamma(r). \quad (13.2)$$

Here, r is the distance between the mesh point and the mesh boundary and the coefficient function is prescribed in a way that makes the coefficient decrease with the distance from the boundary, in order to reduce the displacement.

The solution of the equation 13.1 can be approximated using the FVM in OpenFOAM. In that case, the fields solved for are of course cell centered, with the boundary fields stored at face centers. In order to calculate the displacements in the mesh points (cell corner points) using the FVM, an interpolation from cell centered values to the mesh points needs to be performed. Alternatively, in foam-extend, solution to the equation 13.1, can be approximated using the Finite Element Method (FEM).

Here, a brief design overview of the implementation of mesh deformation within the framework of the FVM in OpenFOAM is provided.

The mesh deformation which involves a solution of the Laplace equation is implemented by `dynamicMotionSolverFvMesh`, shown in figure 13.3. The motion of the moving body is described by the motion of its boundary, which is done by assigning a motion boundary condition to the corresponding part of the body boundary mesh. The motion boundary condition can be given as an explicit function of velocity or displacement, or it can be calculated based upon external data. For example, a data driven mesh motion boundary condition may use data computed as the solution of solid body motion driven by fluid forces integrated on the

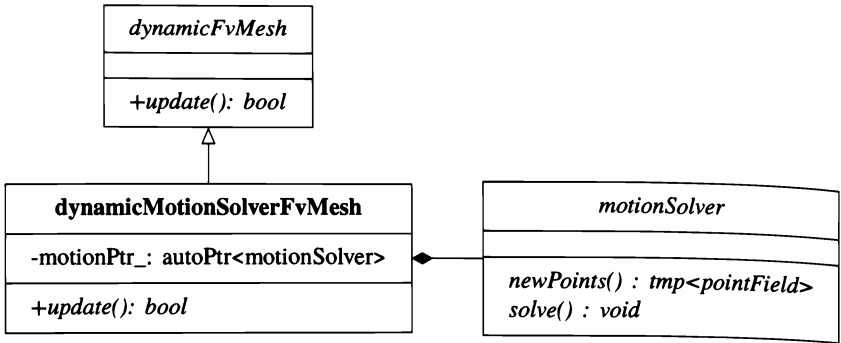


Figure 13.3: Mesh motion implemented by dynamicMotionSolverFvMesh class.

body boundary. In case of a slightly moving body, located inside a large domain, the displacement of outer patches is usually set to a fixed zero value displacement boundary condition, that basically fixes all the points on these domain boundaries in space.

Once the boundary motion is prescribed with a boundary condition for the displacement (velocity) field, the motion solver takes over the responsibility of solving for the motion field. The motionSolver is composited within the dynamicMotionSolverFvMesh, and is implemented as an abstract base class, to allow different approaches to solving for the motion field (e.g. finite volume equation solution, or algebraic interpolation of displacement). The concept of the motion solver is required to generate new mesh points, and nothing else - motion of the mesh points is then further delegated to the parent fvMeshClass:

```

bool Foam::dynamicMotionSolverFvMesh::update()
{
    fvMesh::movePoints(motionPtr_->newPoints());

    if (foundObject<volVectorField>("U"))
    {
        volVectorField& U =
            const_cast<volVectorField&>(lookupObject<volVectorField>("U"));
        U.correctBoundaryConditions();
    }

    return true;
}

```



and the motion solver (`motionPtr_`) is only required to generate new mesh points with the `newPoints` member function:

```
Foam::tmp<Foam::pointField> Foam::motionSolver::newPoints()
{
    solve();
    return curPoints();
}
```

Invoking `newPoints` causes the solution of the mesh deformation to be executed, which in turn modifies the mesh points. The solution process implemented by `solve` will be different depending on which numerical method is used for the solution of the Laplace equation for the displacement, or the chosen interpolation method. There are different motion solver interfaces available, but describing the entire the class structure of this part of the library and the corresponding class interactions are outside of the scope of this book.

Here a brief overview of a Finite Volume based Laplacian mesh motion solver for the displacement field is provided, namely the `displacement-LaplacianFvMotionSolver` whose usage is described in the following section. The two most important classes that interact with the Laplacian finite volume mesh motion solver are shown in figure 13.4. The role of the `displacementMotionSolver` is to encapsulate the modification of the mesh that comes from the topological changes. This allows the displacement motion solvers to be coupled with a topological change of the mesh, as long as the topological change results with a relational map that connects the new topologically modified mesh and the old mesh. More information on topological changes of the mesh can be found in the following section. The `displacementMotionSolver` encapsulates the displacement fields necessary when using mesh deformation, and provides access to them to all the clients, e.g. the displacement solvers. The `motionDiffusivity` is a Strategy (i.e. an OpenFOAM model) used to compute a variable diffusivity coefficient, given by equation 13.2. This spatially variable field can then be used to either scale the displacement which has been propagated to the internal parts of the mesh by interpolation, or it is used as the coefficient in the diffusion equation 13.1. A new function for the motion diffusivity can be developed easily: a class that inherits from `motionDiffusivity` and registers itself to its RTS table is automatically used by the mesh motion framework.

WARNING

Although we have omitted the full description involving all classes that take part in all available mesh motion algorithms, the diagrams and descriptions shown in this section cover the most important implementation aspects relevant for extending mesh motion.

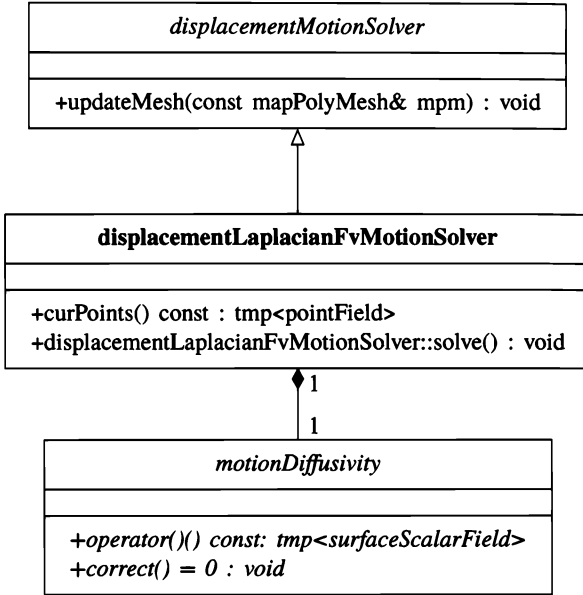


Figure 13.4: Laplacian finite volume based motion solver class diagram.

The Laplacian finite volume mesh motion solver implemented by `displacementLaplacianFvMotionSolver`, shown in figure 13.4, solves the diffusion equation for the cell centered displacement field in the `solve` member function:

```

diffusivityPtr_>correct();
pointDisplacement_.boundaryField().updateCoeffs();

Foam::solve
(
    fvm::laplacian
    (
        diffusivityPtr_>operator>(),
        cellDisplacement_,
        "laplacian(diffusivity,cellDisplacement)"
    )
)

```



```
    )  
};
```

and the point displacements are then interpolated using Inverse Distance Weighted (IDW) interpolation from the cell centers, to the cell corner points (mesh points) within the `curPoints` member function:

```
volPointInterpolation::New(fvMesh_).interpolate  
(  
    cellDisplacement_,  
    pointDisplacement_  
);
```

TIP

A new IDW interpolation object is allocated each time when the `curPoints` is executed - when the mesh is deformed using the FVM based motion solver.

Creating a new IDW interpolation object each time step is necessary as the inversed distance interpolation weights change when a relative displacement exists between mesh points. Since the mesh motion involves mesh deformation with variable relative displacements between all mesh points, caching the IDW interpolation weights to increase efficiency of the calculation is not possible.

13.1.2 Topological Changes

Changing the topology of the mesh involves modifying its topological information: adding and deleting mesh elements (cells, faces, edges, points) and updating all the data structures that describe the mutual connectivity between mesh elements (e.g. the edge-cells connectivity list). Applying topological changes to the mesh is usually motivated either by increasing the accuracy in the regions of the domain where large gradients are present, or by modeling dynamical systems whose simulation domains experience extreme changes in shapes and sizes. Mesh deformation may cause severe degradation of mesh quality as the angles between cell faces and ratios of neighboring cell sizes can be severely modified. As a result of a severe mesh deformation, the underlying FVM may lose accuracy on such meshes, even in those regions where accuracy is important. Accuracy is lost because the cells get distorted in a way that introduces

interpolation errors because: large ratios of neighboring cell sizes appear and face angle distortion can result in the increased non-orthogonality of the mesh.

To avoid such problems, the mesh deformation may be coupled with topological changes of the mesh, which results with both an accurate and efficient dynamic mesh engine. An example of such an advanced implementation is the `dynamicTopoFvMesh`¹ library (S. Menon, Rothstein, Schmidt, and Tukovic 2008, Sandeep Menon 2011, S. Menon and Schmidt 2011).

In order to support building complex topological changes of the mesh, topological changes in OpenFOAM are designed as simple operations which can then be agglomerated into more complex ones. The agglomerated operations can then be used to build specialized dynamic mesh classes.

Dynamic meshes that deal with changing of the mesh topology usually are more specialized than the mesh motion classes, because changing the mesh topology during runtime is significantly more difficult. The difficulty lies in operations on the mesh topology, as well as modification of the field values to account for the topological change. Most of the classes that perform topological changes are tailored towards one specific task. The `dynamicFvMesh` library, located at `$FOAM_SRC/dynamicFvMesh` contains solely more general purpose topological changers, such as `dynamicRefineFvMesh`. This dynamic mesh implements dynamic and local refinement of a hexahedral mesh, based on a value interval for the refinement criterion (e.g. the pressure gradient). The user defines the refinement criterion value interval and the cells that contain field values lie inside of a user prescribed interval get dynamically refined, and those that store values outside that interval get unrefined. Local dynamic mesh refinement is useful for problems when e.g. the region in the near vicinity of a moving interface between two fluids needs to be highly resolved and bulk of two fluid phases does not. Using different fields as refinement criteria is possible as well. For example, the refinement criterion can be computed using a more complicated model executed during the simulation in the form of a user developed function object.

¹Can be obtained freely from <https://github.com/smenon/dynamicTopoFvMesh>



More specialized topological changers can be found in `$FOAM_SRC/topoChangerFvMesh` and the `movingConeTopoFvMesh` serves as an example. This dynamic mesh is very problem specific: it is used to model the dynamic simulation of a moving piston within a cylinder. The mesh moves a patch (a part of the mesh boundary) in the direction of the x -axis and adds and removes cell layers where needed in the region where the cells get highly deformed, in order to keep a high enough mesh quality for an accurate numerical simulation. Although this dynamic mesh implements a combination of mesh motion and topological changes of the mesh, the constant linear velocity of the patch is not the part that has the highest demand in terms of programming effort. The ability to develop such a specialized dynamic mesh is a result of separation of abstraction levels: on the lower level single topological operations exist, and on the higher level of abstraction, their agglomeration results in an operation that adds whole cell layers to the mesh. There are different topological changers available in OpenFOAM, and describing all of them and their design background lies outside of the scope of this book. Here we only provide a brief design overview of the `dynamicRefineFvMesh` class.

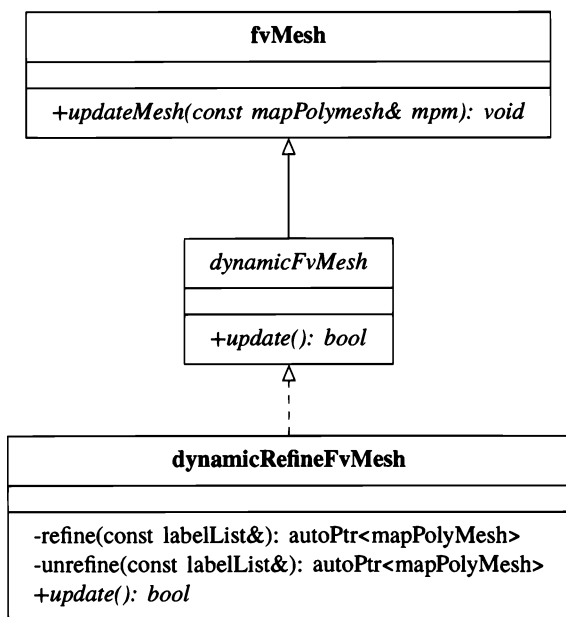


Figure 13.5: Mesh refinement implemented by the `dynamicRefineFvMesh` class.

Figure 13.5 shows important class relationships of the `dynamicRefineFvMesh` class. As other dynamic mesh classes, it implements the `update` member function to conform to the `dynamicFvMesh` interface. The mesh refinement operation involves refinement (splitting) and unrefinement (merging) of cells, depending on the above described refinement criterion. Both operations are implemented by two private member functions `refine` and `unrefine`, respectively. Each time either of the two operations are executed the `dynamicRefineFvMesh` class collaborates with the parent `fvMesh` class in order to update the fields for topological operations.

TIP

The collaboration between the `dynamicRefineFvMesh` and `fvMesh` is achieved by calling the `fvMesh::updateMesh` member function. The mapping of fields may be a complex algorithm to implement. When a new class for topological changes is implemented, it may be easier to re-use this algorithm by adhering to the class relationship shown in figure 13.5



Algorithm 2 Mesh refinement and unrefinement algorithm

```

read the control dictionary for the dynamicRefineFvMesh
if not first time step then
    look up the refinement criterion field
    if number of mesh cells < maxCells then
        select cells to be refined using the refinement criterion (refineCells)
        modify refine cells by adding cells from the refinement layer and
        protected cells
        if cells to be refined > 0 then
            refinementMap = refine(cellsToBeRefined)
            update refineCell (refinementMap)
            updateMesh (refinementMap)
            mark the mesh as changed
        end if
    end if
    select unrefinement points
    if number of unrefinement points > 0 then
        refinementMap = unrefine (unrefinement points)
        updateMesh (refinementMap)
        mark the mesh as changed
    end if
end if

```

The algorithm for mesh refinement/unrefinement is shown, with a reduced level of detail, in algorithm 2. The operations of refinement and unrefinement generate a *topological map* that connects the new topologically modified mesh, with the original mesh. A more detailed description of the refinement algorithm is out of scope for this book, and might be important for those readers that intend to extend or implement their own mesh refinement algorithm.

Splitting of mesh faces results with cells that are polyhedral - the cells are still cubical but have more than six faces. As long as two cells are separated by the same number of sub-faces produced by splitting, the owner-neighbor indirect addressing of the mesh (chapter 1) will work without modification. However, it is important to note that such splitting introduces non-orthogonality, aspect ratio, as well as small skewness errors. Because of this, it is common to provide an *additional layer of refined cells*: the boundary of the refined cell layer should lie in the region with small gradients, where high accuracy is not as important as

within the refined cell layer.

TIP

Adaptive refinement of hexahedral unstructured mesh in OpenFOAM *is not* based on an octree data structure. The mesh topology is changed directly and stored for the new mesh. The same discretization operators are then using the *new mesh topology* to discretize the model equations.

13.2 Usage

Any solver that contains DyMFoam in its name is capable of using any dynamic mesh available, regardless of using topological changes of the mesh or mesh motion. The only requirement that all have in common is the `dynamicMeshDict`, which must be present in the `constant` directory and be configured appropriately.

Two examples are provided in this section and each relates to the mesh motion types described in the previous section: global mesh motion using `solidBodyMotionFvMesh` and mesh deformation based on `dynamicMotionSolverFvMesh`. Using topological mesh changes is not shown in this section as the next section provides another example for that purpose. Both examples utilize the same base mesh, which is a unit-cube immersed in a bigger cubical domain. In both cases, the inner cube performs the same motion, but each example computes the mesh motion differently. The two examples prescribe a linear translational motion and use the `solidBodyMotionFvMesh` and the `dynamicMotionSolverFvMesh` dynamic mesh class, respectively. The example base case is located in the example case repository under `chapter13/unitCubeBase`.

Setting up OpenFOAM cases that use the dynamic mesh capabilities can be a tedious work, especially if the flow solver takes up a lot time during a time step. In order to avoid waiting for a long time, just to check if the case is set up correctly, the tool `moveDynamicMesh` can be used. It performs all steps the solvers make when using dynamic meshes, but without the expensive flow solution. Hence, only `mesh.update()` is executed inside the time loop, which triggers the mesh modifications performed by the runtime selected dynamic mesh class. This works well, as long as the motion is not dependent on any data that comes from the



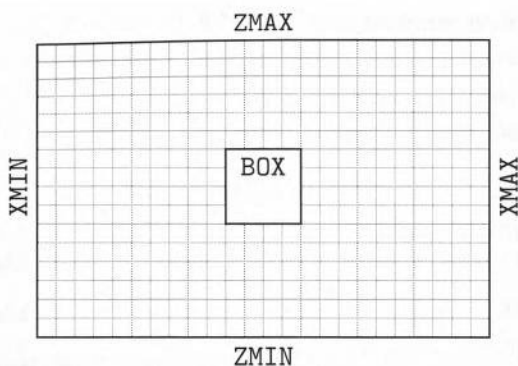


Figure 13.6: Sketch of the geometry in the $x - z$ plane, used in the `unitCubeBase` case.

flow simulation.

TIP

When a simulation case is to be extended to include dynamic mesh operations, using `moveDynamicMesh` application is recommended because of shorter execution time compared to the flow solver.

13.2.1 Global Mesh Motion

As outlined in the beginning of this section, global mesh motion can be achieved by using the `solidBodyMotionFvMesh` in conjunction with a `solidBodyMotionFunction` of any choice. To achieve an linear translational motion, `linearMotion` is used in this example. Similar to all other tutorial cases described in this book, the first step is to copy the tutorial to a location where it can be edited safely. To do so, change into the example case repository and locate the `chapter13` directory:

```
?> cp -r chapter13/unitCubeBase $WM_PROJECT_USER_DIR/run
?> cd $WM_PROJECT_USER_DIR/run/unitCubeBase
```

The `dynamicMeshDict` is the only configuration file that needs to be adjusted, if the mesh motion needs to be changed. As can be seen from the

following excerpt, it employs the `solidBodyMotionFvMesh` to define the mesh motion class and the `linearMotion` defines the motion function.

```
dynamicFvMesh    solidBodyMotionFvMesh;

solidBodyMotionFvMeshCoeffs
{
    solidBodyMotionFunction    linearMotion;
    linearMotionCoeffs
    {
        velocity              (1 0 0);
    }
}
```

The above dictionary instructs `solidBodyMotionFvMesh` to obtain the transformation from `linearMotion` and apply it to all mesh points, rather than a zone. Before being able to test the motion, the mesh needs to be generated. Therefore execute the following two steps, but be aware that the second step will generate time step directories in the case folder:

```
?> blockMesh
?> moveDynamicMesh
```

The second call executes `moveDynamicMesh` which in turn takes care of the mesh motion and provides several lines of output to the screen per time step. Depending on the configuration of the `controlDict`, the time step directories are generated at various frequencies. With the current case configuration in the example case repository, the data is written at every 0.05 seconds and can be inspected using `paraView`. Chapter 4 provides description on how to use `paraView` for visualizing OpenFOAM results.

13.2.2 Mesh Deformation

In comparison to the global mesh motion example presented in the previous section, mesh morphing is a little more demanding, in terms of configuration efforts that have to be spent. Not only the `constant/dynamicMeshDict` is responsible for the mesh motion, but also a new field in the `0` directory and an additional solver in `system/fvSolution` need to be added. The last two entries depend on the *type* of the mesh motion solver, and they have been briefly addressed in section 13.1. For this



example, a displacement based mesh motion solver is selected. Therefore, the added new field is named `pointDisplacement`, which is a `pointVectorField` and the respective boundary conditions need to be set for this field:

```
dimensions      [0 1 0 0 0 0];

internalField    uniform (0 0 0);

boundaryField
{
    "(XMIN|XMAX|YMIN|YMAX|ZMIN|ZMAX)"
    {
        type      fixedValue;
        value      uniform (0.75 0 0);
    }
    BOX
    {
        type      fixedValue;
        value      uniform (1 0 0);
    }
}
```

Rather than defining a motion only for the box itself, a velocity is assigned to the outer boundaries as well, in order to illustrate the capabilities of the `dynamicMotionSolverFvMesh`. The box has the same velocity as for the previous tutorial, but it is reduced to 0.75 for the remaining patches. Of course, this setup has a practically limited time of execution, as the box will compress the cells too much with its motion. Still, this setup serves as an example of how various motions can be assigned to different boundaries.

The `dynamicMeshDict` of the previous example needs to be changed as well and should look like the following:

```
dynamicFvMesh      dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver             displacementLaplacian;

displacementLaplacianCoeffs
{
    diffusivity      inverseDistance (BOX);
}
```

As already mentioned earlier, a new solver entry in `system/fvSolution` is required. A GAMG type solver with a GaussSeidel smoother is selected for this purpose. The following entry must be added to the `solvers` sub-dictionary of `fvSolution`:

```
cellDisplacement
{
    solver      GAMG;
    tolerance   1e-5;
    relTol      0;
    smoother    GaussSeidel;
    cacheAgglomeration true;
    nCellsInCoarsestLevel 10;
    agglomerator faceAreaPair;
    mergeLevels 1;
}
```

With the addition of the `0/pointDisplacement` boundary file, the above entry into `fvSolution` and the described changes to the `dynamicMeshDict`, the `unitCubeBase` example can be executed again.

```
?> rm -rf 0.* [1-9]*
?> moveDynamicMesh
```

13.3 Development

This section clarifies the development of new dynamic mesh classes and how standard solvers can be extended to make use of the dynamic mesh capabilities of OpenFOAM. Before going into detail of how new dynamic meshes can be developed, a standard solver is extended with dynamic mesh capabilities.

13.3.1 Adding Dynamic Mesh to a Solver

Extending an existing solver to use the dynamic mesh features of OpenFOAM is a fairly straightforward process. For this example, the `scalarTransportFoam` solver is selected as a solver that requires extension in order to support dynamic mesh handling. As usual, the first step is to create a copy of the original `scalarTransportFoam` application folder in an arbitrary directory on your computer, change into that directory and remove the existing dependency files and the like:



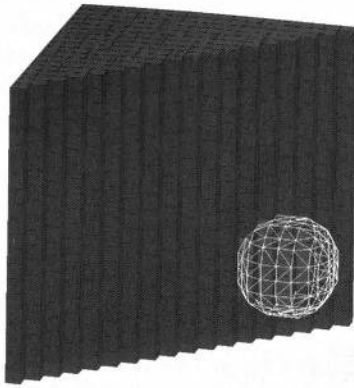


Figure 13.7: Test case for scalar transport with mesh refinement. Red cells are initialized with the value 100, and blue cells with value 0, the wireframe sphere is an iso-surface with the iso-value of 50.

```
?> cp -r $FOAM_SOLVERS/basic/scalarTransportFoam \
    $WM_PROJECT_USER_DIR/applications/scalarTransportDyMSolver
?> cd $WM_PROJECT_USER_DIR/applications/scalarTransportDyMSolver
?> rmdepall && wclean
```

For this example, we have used the standard applications user directory in the OpenFOAM installation directory hierarchy. To keep things organized, some files should be renamed and altered. Firstly rename `scalarTransportFoam.C` to `scalarTransportDyMSolver.C` and secondly replace all occurrences of `scalarTransportFoam` in `Make/files` by `scalarTransportDyMSolver`. It is important to compile the solver executable into `$FOAM_USER_APPBIN`, rather than the standard `$FOAM_APPBIN`. This needs to be changed accordingly in `Make/files`. After all these changes are applied, the existing code should to be compiled and tested. The results must be the same as with standard `scalarTransportFoam`, and it can be tested on the simulation case `chapter13/scalarTransportAutoRefine` in the example case repository. Figure 13.7 shows the initial conditions and the domain geometry for the test case used with this example. The test case consists of a cubical domain with an initial field preset as a sphere in the corner of the domain, and a constant velocity field which is used to transport the field in the direction of the spatial diagonal.

Having verified that `scalarTransportDyMSolver` works as expected, the dynamic mesh capabilities must be implemented. To do so, `scalarTransportDyMSolver.C` must be opened in a text editor and `dynamicMesh.H` must be included after `simpleControl.H`:

```
#include "fvCFD.H"
#include "fvIOoptionList.H"
#include "simpleControl.H"
#include "dynamicFvMesh.H"
```

In the main function `createDynamicFvMesh.H` rather than `createMesh.H` has to be included, in order to deal with dynamic meshes properly:

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
```

A closer look at the sources of `scalarTransportFoam` reveals that there is no call to `mesh.update()`, which is responsible for executing the dynamic mesh functions. This must hence be added at the end of the time loop:

```
    mesh.update();
    runTime.write();
}

Info<< "End\n" << endl;

return 0;
```

In order to use the dynamic mesh, the included header files need to be found by the compiler and the dynamic library code needs to be found by the dynamic linker. To allow both, the `Make/options` file needs to be modified, first a line is added to account for the included `dynamicFvMesh.H` file

```
-I$(LIB_SRC)/sampling/lnInclude \
-I$(LIB_SRC)/dynamicFvMesh/lnInclude
```



and the application is linked against the `libdynamicFvMesh` dynamically loadable library, by adding the appropriate line to the list of libraries

```
EXE_LIBS = \
    -lfiniteVolume \
    -ldynamicFvMesh \
```

The `dynamicRefineFvMesh::update` member function will correct the volumetric flux² values for the new faces generated by the mesh refinement procedure. This process is named *flux mapping* and can be governed by the user by modifying the *flux mapping table* placed in the `constant/dynamicMeshDict`.³ However, the mapped volumetric fluxes are usually used as a good enough initial guess for the flow solution algorithm, that enforces volume conservative volumetric flux values. For a scalar transport equation in this example, there no flow solution algorithm is executed, as the field is advected passively, using a *constant prescribed velocity*. The refinement procedure makes the velocity non-constant indirectly, by modifying the flux values to account for new faces in the mesh.

If the solver is to be compiled and run at this point, with the default settings provided in the example simulation case, the solution becomes numerically unbounded. To solve this problem, we insert the last line of code into the solver application, that is used to mimick the presence of a flow solution algorithm, guaranteeing the volume conservative volumetric flux values. The line is inserted just below the call to the `update` member function:

```
mesh.update();
phi = fvc::interpolate(U) & mesh.Sf();
```

The code is now ready to be compiled - by executing `wmake` in the source directory. Newly compiled solver is ready to be tested on the same case as before: `scalarTransportAutoRefine` in the example case repository. Figure 13.8 shows the transported spherical scalar field `T` initialized as shown in figure 13.7, with enabled dynamical adaptive mesh refinement.

²The importance of the volumetric flux in the transport of a physical property using the finite volume method is described in detail in chapter 1.

³More information on the mesh refinement algorithm can be found in section 13.1.

The diffusive transport of T causes the jump to be smeared up as it gets advected, but still it can be observed that the mesh refinement is dynamic and is following the field transport.

WARNING

By refining the mesh, new cell faces are created, which in turn requires the modification of the volumetric flux field. The volume conservation, numerical boundedness as well as solution stability depend strongly on the volumetric flux values.

EXERCISE

The refinement criterion used for this test case refines uniformly the interior of the transported sphere. An interesting exercise would be to apply a refinement criterion that would follow the jump in values of T . This would decrease computational costs and increase the accuracy of the diffusive transport.

13.3.2 Combining two Dynamic Mesh Classes

In this subsection, the development of a new dynamic mesh class is described. The aim is to combine two different dynamic mesh types into a unified dynamic mesh. The new dynamic mesh class combines solid body mesh motion and adaptive hexahedral mesh refinement, both are described in section 13.1.

Operations related to the role of the mesh in the unstructured FVM increase severely the level of complexity, when it comes to extending the dynamic mesh handling capability. From the software design side, the mesh classes in OpenFOAM take over multiple responsibilities and have a complex collaboration structure with other classes. For example, the dynamic mesh classes collaborate strongly both with the simulation time and the fields involved in the simulation. This includes updating the fields when the mesh itself is changed. Both issues raise the level of complexity when it comes to extending dynamic mesh handling.

Figure 13.9 shows the relevant part of the class hierarchy used for mesh motion and refinement. The `update` member function used for updating

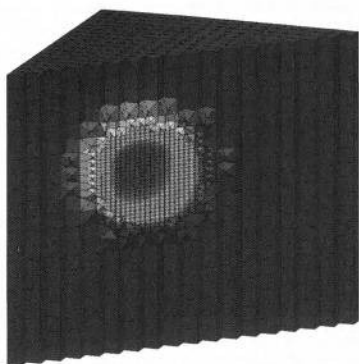


Figure 13.8: Scalar field transported with enabled local dynamic mesh refinement.

the mesh for motion or topological changes is emphasized. Additionally the relationship between the `fvMesh` concrete class, the `dynamicFvMesh` interface and other dynamic mesh classes that implement the `dynamicFvMesh` interface is depicted in that figure. All other properties are neglected, for the sake of simplicity.

The `fvMesh` is a concrete class that implements necessary geometrical information of the mesh, essential to the unstructured FVM. All dynamical operations in OpenFOAM are abstracted into the `dynamicFvMesh` interface and implemented there. Note that `dynamicFvMesh` inherits from `fvMesh`, which is a concrete class on it's own. As a result, each time an object of a class that implements the `dynamicFvMesh` interface is instantiated, a complete `fvMesh` is created as well. This is important to keep in mind, for developing new dynamic mesh classes.

Functionalities of classes are usually extended, either by inheritance or composition, or some combination of those two. Combining multiple functionalities such as mesh motion and refinement using multiple inheritance is excluded as shown by the structure outlined in figure 13.9. A single new dynamic mesh object will contain *two instances* of `fvMesh`, which would incur huge computational and semantical costs for absolutely no benefit. Composing an object of `dynamicFvMesh` will result in the same situation. This leads to conclusion that the dynamic mesh classes are not meant to be extended by using inheritance and/or composition,

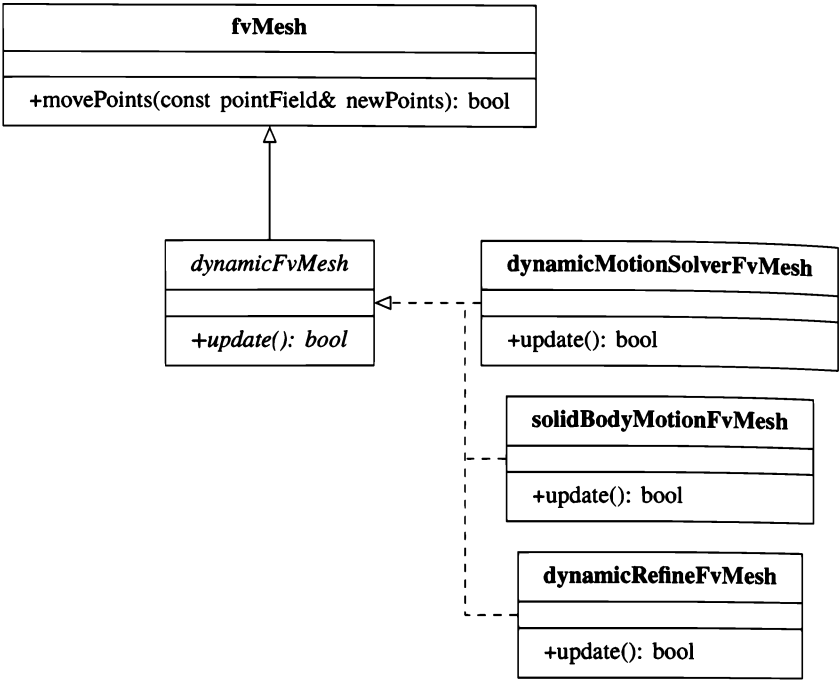


Figure 13.9: Simplified class diagram for the dynamic mesh classes

WARNING

Each object of the `dynamicFvMesh` interface class is also an object of the `fvMesh` concrete class - a fully functional finite volume mesh. Hence, using multiple inheritance with classes derived from `dynamicFvMesh` would cause multiple meshes to be allocated.

by design.

Accepting two instances of the same class being instantiated, due to such design constraints is usually acceptable if either one of the following requirements is matched:

- the class in question has a small memory footprint,
- the objects of the final class with extended functionality are not going to be generated in large numbers.



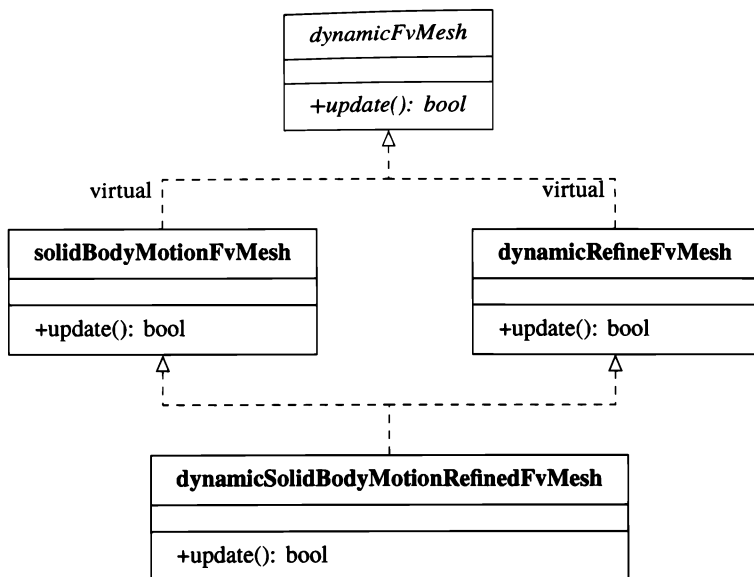


Figure 13.10: Combining dynamic mesh classes using multiple inheritance results with a diamond inheritance structure.

WARNING

The solutions we present in this section represent our own conclusions on how the dynamic mesh classes are to be combined in OpenFOAM.

This is not a general advice and has to be used with high caution. Duplicating objects is one of the most often encountered problems in computational efficiency, but profiling the code is the only way to judge if this issue causes a computational bottleneck. Yet another problem is the consistency in data duplication. A CFD simulation expects a single set of points, cells, boundaries, etc. to be stored in the mesh and be existent in memory. Duplication of that data could cause inconsistencies in the computation, if the data is not synchronized after modifications. The computational meshes have millions of points, volumes, area normal vectors and similar data. Therefore it is safe to say that duplicating an entire mesh for no obvious gain makes no sense. When composition of dynamic mesh classes is used, there is no way around this problem, though.

On the other hand, if multiple inheritance would be used to create a new dynamic mesh class from the `solidBodyMotionFvMesh` and `dynamicRefineFvMesh`, a "diamond inheritance" as shown in figure 13.10 would be the result. In this case, calling the constructor of `fvMesh` twice can be avoided, by employing *virtual inheritance* for the dynamic mesh classes:

```
class solidBodyMotionFvMesh
:
    public virtual dynamicFvMesh
{
    ...

class dynamicRefineFvMesh
:
    public virtual dynamicFvMesh
{
    ...
```

The multiplied construction of the `dynamicFvMesh` object is then avoided, by calling the virtual base class constructor explicitly in the inherited class:

```
class dynamicSolidBodyMotionRefinedFvMesh
:
    public solidBodyMotionFvMesh,
    public dynamicRefineFvMesh
{
    public:
        dynamicSolidBodyMotionRefinedFvMesh(arguments)
        :
            dynamicFvMesh(arguments),
            solidBodyMotionFvMesh(arguments),
            dynamicRefineFvMesh(arguments)
```

Even if the approach of virtual inheritance might help in this case, the OpenFOAM library would have to be changed, in order to achieve this solution.

The considerations described in this chapter show how we have approached this example problem and how we tried to avoid code du-



plication by re-using as much code as possible and not imposing any changes to the existing library code at the same time. The worst thing that could be done in such a situation is to simply copy code from two classes into one. At first glance, the programming speed makes this idea quite appealing. However, with complex operations such as mesh changes, it will surely happen that bugs are found and resolved upstream by the library maintainers. In that case, each time the library is updated upstream, changes need to be merged manually. This leads to a lot of repeated work for no visible gain.

WARNING

Always consider the proven software design approaches to avoid code duplication, modification of existing code and maximize code re-use. Copying and pasting source code is the worst option to take.

With the goal of this example being the combination of hex mesh refinement with solid body mesh motion, checking the class structures of `dynamicRefineFvMesh` and `solidBodyMotionFvMesh` is the next step. The focus is on discovering encapsulated sub-algorithms and possibly design patterns, that can be reused. Examining the source code of both classes leads to conclusion that `solidBodyMotionFvMesh` is less complex to partially re-build than `dynamicRefineFvMesh`. For this reason, the new class will inherit from `dynamicRefineFvMesh` and will re-use elements of `solidBodyMotionFvMesh`.

Figure 13.11 shows the attributes of `solidBodyMotionFvMesh` that are going to be re-used. When the mesh is moved, a new set of points is created by displacing the old points. The mesh modification is taken care of by the responsible part of the update member function of `solidBodyMotionFvMesh`:

```
fvMesh::movePoints
(
    transform
    (
        SBMFPtr_().transformation(),
        undisplacedPoints_
    )
);
```

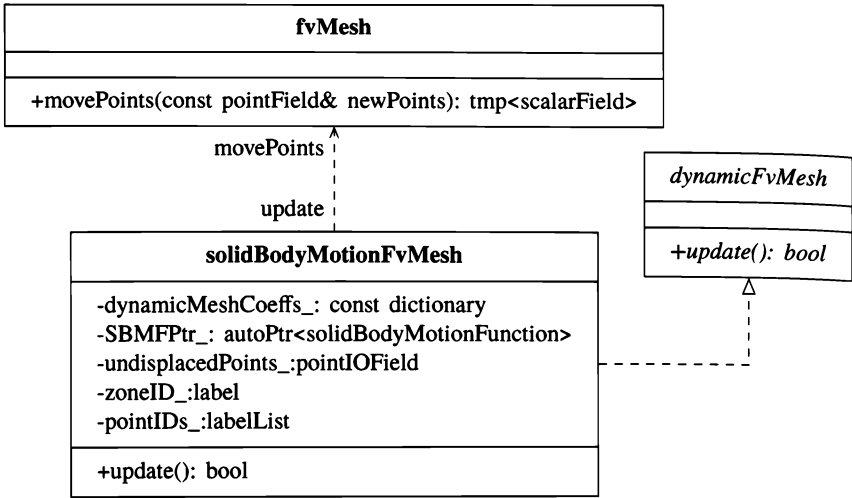


Figure 13.11: Class diagram of `solidBodyMotionFvMesh`.

The `solidBodyMotionFvMesh` class delegates the generation of the displaced points to the `solidBodyMotionFunction` and the actual operation of displacing the mesh points to the `fvMesh` class. This association is shown in figure 13.11. The remaining attributes shown in figure 13.11 and the class implementation are responsible for executing the motion on a sub-set of mesh cells. As this example moves the entire mesh, those parts of the code do not have to be used at all.

TIP

The `solidBodyMotionFunction` is an application of a Strategy design pattern, described in section 13.1.1, which makes it very straightforward to re-use its functionality.

To sum up the conclusions on the design of the new `solidBodyMotionRefinedFvMesh` class:

- multiple inheritance is not to be used, as it requires library modification to prevent multiple calls to the `fvMesh` constructor,
- composition cannot be used as it cannot avoid multiple calls to the `fvMesh` constructor,
- the `dynamicRefineFvMesh` class is more complex, so it is chosen



as the base class of the single inheritance extension,

- mesh motion is to be added using composition on the `pointField` and the `solidBodyMotionFunction` attributes of the `solidBodyMotionFvMesh` class.

At this point the information should suffice to start with the actual implementation of a new dynamic mesh class that combines mesh refinement and solid body motion. The new library can be named in whatever way, for this example `dynamicSolidBodyMotionRefinedFvMesh` is chosen. First, a directory with this name needs to be created:

```
?> mkdir dynamicSolidBodyMotionRefinedFvMesh
?> cd dynamicSolidBodyMotionRefinedFvMesh
```

The class files can be generated using the `foamNew` script:

```
?> foamNew source H dynamicSolidBodyMotionRefinedFvMesh
?> foamNew source C dynamicSolidBodyMotionRefinedFvMesh
```

Which creates the implementation stubs for the class. These files need to be cleaned up of lines that are of no use for this example. In the following code snippets the file headers are not shown, since they do not bring any helpful information to the tutorial description. Note however that they should never be removed, as removal of headers breaches the GNU Public License, under which the code has been released to the general public. The relevant part of the cleaned up `dynamicSolidBodyMotionRefinedFvMesh.H` file should look like this:

```
namespace Foam
{

class dynamicSolidBodyMotionRefinedFvMesh
{
    // Private data

public:

    // Static data members

    // Constructors

    //- Construct null
    dynamicSolidBodyMotionRefinedFvMesh();
```

```
//- Destructor
~dynamicSolidBodyMotionRefinedFvMesh();

// Member Functions

};

} // End namespace Foam
```

The implementation file `dynamicSolidBodyMotionRefinedFvMesh.C` should look like the following:

```
#include "dynamicSolidBodyMotionRefinedFvMesh.H"

// ***** Static Data Members ***** //

// ***** Constructors ***** //

Foam::dynamicSolidBodyMotionRefinedFvMesh::
dynamicSolidBodyMotionRefinedFvMesh()
{}

// ***** Destructor ***** //

Foam::dynamicSolidBodyMotionRefinedFvMesh::
~dynamicSolidBodyMotionRefinedFvMesh()
{}


```

Now that the minimal classes are created, the library needs to be compiled and tested. First, the Make directory is created with files and options files required by the `wmake` build system:

```
?> mkdir Make
?> cd Make
?> touch files
?> touch options
```

The files must contain the following entries:

```
dynamicSolidBodyMotionRefinedFvMesh.C

LIB = $(FOAM_USER_LIBBIN)/libsolidBodyMotionRefined
```

And the options file must look as follows:




```

EXE_INC = \
-I$(LIB_SRC)/finiteVolume/lnInclude \
-I$(LIB_SRC)/dynamicMesh/lnInclude \
-I$(LIB_SRC)/dynamicFvMesh/lnInclude

EXE_LIBS = \
-lfiniteVolume \
-ldynamicMesh \
-ldynamicFvMesh

```

At this point the library can be compiled by executing `wmake` in the `dynamicSolidBodyMotionRefinedFvMesh` directory. Having successfully compiled the library, it can be tested. The description of this tutorial is a step-by step process. As each of the design conclusions described earlier is addressed, the library is tested continuously. Compilation or runtime errors are dealt with, as they happen to occur. Nevertheless, using the *tests first* approach for this example is beneficial, as it reduces the complexity of the errors.

The test case for the combined solid body mesh motion and refinement is prepared and can be found in the example repository in the subdirectory `chapter13/movingRefinedMesh`. The test itself is very simple: it consists of linear translation of a simple shaped domain, which is filled with two fluid phases. The solver `interDyMFoam` will be used to test the new library. For reasons that will be addressed at a later point in the section, the layer of cells that hold the interface between two phases has been initially refined. Figure 13.12 shows the initial configuration and preprocessed `alpha1` field for the test case.

To test the library, the combined mesh motion and refinement library needs to be switched off at runtime. To do so, the `system/controlDict` needs to be edited as follows: All lines responsible for inclusion of other libraries as well as function object invocation need to be commented out.

```

//libs (
//  "libofBookDynamicFvMesh.so"
//  "libutilityFunctionObjects.so"
//);
//
//functions
//{
//  courantNo
//  {

```

```
//      type CourantNo;  
//      phiName phi;  
//      rhoName rho;  
//      outputControl outputTime;  
//  }  
//}
```

As a next step, the "libsolidBodyMotionRefined" library must be linked to the solver during runtime:

```
libs("libsolidBodyMotionRefined.so");
```

To finally test the library, run the `interDyMFoam` solver within the `movingRefinedMesh` simulation case directory. The following error should appear:

```
--> FOAM FATAL ERROR:  
Unknown dynamicFvMesh type dynamicSolidBodyMotionRefinedFvMesh  
  
Valid dynamicFvMesh types are :  
  
8  
(  
dynamicInkJetFvMesh  
dynamicMotionSolverFvMesh  
dynamicRefineFvMesh  
movingConeTopoFvMesh  
multiSolidBodyMotionFvMesh  
rawTopoChangerFvMesh  
solidBodyMotionFvMesh  
staticFvMesh  
)
```

In case an error of the following type appears:

```
--> FOAM Warning :  
  From function dlOpen(const fileName&, const bool)  
  in file POSIX.C at line 1179  
  dlopen error : libsomething.so:  
  cannot open shared object file: No such file or directory  
--> FOAM Warning :  
  From function dlLibraryTable::open(const fileName&, const bool)  
  in file db/dynamicLibrary/dlLibraryTable/dlLibraryTable.C at line 99  
  could not load "libsomething.so"  
Create mesh for time = 0
```

It indicates that there is an error in the library lookup. The library name



"libsomething" is used as an example of a library that is not registered to the RTS table of the `dynamicFvMesh`. Whenever this kind of error occurs, either the name of the library to be loaded is misspelled, or the library has not compiled properly.

To correct the error of the unknown `dynamicFvMesh` type, the conclusions on the implementation addressed above need to be transferred into the new library code. The first conclusion that reflects on the code is inheriting from the `dynamicRefineFvMesh`. This is applied in the header file:

```
#include "dynamicRefineFvMesh.H"

// *****

namespace Foam
{

class dynamicSolidBodyMotionRefinedFvMesh
:
    public dynamicRefineFvMesh
{
    // Private data
```

Since the new class is placed in a class hierarchy with virtual functions, a virtual destructor has to be declared:

```
//- Destructor
virtual ~dynamicSolidBodyMotionRefinedFvMesh();
```

In order to resolve the first observed error, further steps are necessary. The new class needs to be added to the RTS table of the `dynamicFvMesh` class. RTS in OpenFOAM relies on a string (`Foam::word`) based type system. Each class defines a global static string variable, the type name. Usually, the root class of the hierarchy stores a static public attribute in the form of a (hashed) table, that relates strings (type names) to pointers to class constructors. As the hash table is declared as a static publicly accessible object, the derived classes can modify its state. They can add their type name, and a pointer to their constructor, to that table. As a result, the base class can then simply browse through the table and get a handle (pointer) to the constructor of every derived class. The RTS itself

deserves separate attention and it is outside of the scope of this book.⁴ Proper usage of the RTS system in OpenFOAM is shown in various examples throughout this book, though.

To continue building the new dynamic mesh class, the type name must be declared in the header:

```
// Static data members
TypeName ("dynamicSolidBodyMotionRefinedFvMesh");
```

To use the `TypeName` macro, the header file containing its definition must be included in the header of the library:

```
#include "dynamicRefineFvMesh.H"
#include "typeInfo.H" // This line is newly added.
```

Once the static type name variable and the debug switch are declared, it needs to be defined. This is done by means of the macro in the `dynamicSolidBodyMotionRefinedFvMesh.C` file:

```
namespace Foam {
    defineTypeNameAndDebug(dynamicSolidBodyMotionRefinedFvMesh, 0);
}
```

At this point, the interface of the new class has to be adjusted, in order to follow the one prescribed by the parent `dynamicRefineFvMesh`. Only one constructor is provided, for the derived new dynamic mesh class in this example. The previously defined empty constructor is to be removed, and replaced with the following constructor:

```
// Constructors

//- Construct from objectRegistry, and read/write options
explicit dynamicSolidBodyMotionRefinedFvMesh(const IObject& io);
```

It has to be defined accordingly in the `.C` file:

```
dynamicSolidBodyMotionRefinedFvMesh(const IObject& io)
{
```

⁴There is a description of the RTS mechanism in OpenFOAM available on the OpenFOAM Wiki page.



```
dynamicRefineFvMesh(io)
{}
```

In the current state, the class still has no private attributes. Hence the definition of the constructor will be modified as private attributes are added, in order to initialize them properly and construct a valid object. At this point the new dynamic mesh class inherits from `dynamicRefineFvMesh` and has thus a similar functionality. In order to make the class usable, it needs to conform to the `dynamicFvMesh` interface, by declaring the `update` member function:

```
// Member Functions

// - Update the mesh for both mesh motion and topology change
virtual bool update();
```

The corresponding definition in the `.C` file should look like the following:

```
bool Foam::dynamicSolidBodyMotionRefinedFvMesh::update()
{
    dynamicRefineFvMesh::update();

    return true;
}
```

The `update` member function of the `dynamicRefineFvMesh` class is being called explicitly. This is because C++ shadows the names of the parent classes when inheritance is used. As soon as `update` member function is declared in the new class, it has hidden the same function of the parent class. By calling this member function, the new class behaves exactly as the parent class. To allow the use of the new dynamic mesh class, its constructor needs to be added to the RTS table of the `dynamicFvMesh`. Otherwise the solver will not be able to select and construct the object the new dynamic mesh class, depending on the type name provided in the `constant/dynamicMeshDict` file. In order to do that, the following macro in the `.C` file must be called:

```
addToRunTimeSelectionTable
(
    dynamicFvMesh,
    dynamicSolidBodyMotionRefinedFvMesh,
```

```
IObject  
);
```

The corresponding macro needs to be added by including the appropriate file:

```
#include "addToRunTimeSelectionTable.H"
```

At this point, the library should be compiled again, to check if the changes the were applied to the the new dynamic mesh class are correct. The new class should now be selectable by adding the type name entry in the `system/dynamicMeshDict` file of the `movingRefinedMesh` simulation case:

```
dynamicFvMesh dynamicSolidBodyMotionRefinedFvMesh;
```

Running `interDyMFoam` should cause no problems at this point and the new class should behave just like the `dynamicRefineFvMesh` class. Since the refinement of the mesh is running now, due to being inherited from `dynamicRefineFvMesh`, solely the motion of the mesh points needs to be added. This is done by following the design conclusions and by adding private attributes and related functionality of the `solidBodyMotionFvMesh` class. The solid body motion must be added to the class and select the particular solid body motion function independently. The new mesh points that are created by transforming the original mesh points with parameters of the motion function. First the new private attributes are added to the class declaration:

```
//- Dictionary of solid body motion control parameters  
const dictionary motionCoeffs_;
```



```
//- The motion control function  
autoPtr<solidBodyMotionFunction> SBMFPtr_;
```



```
//- The reference points which are transformed  
pointIOField undisplacedPoints_;
```

Compared to the `solidBodyMotionFvMesh` class, this set of attributes is somewhat simplified. It doesn't account for motion of a cell sub-set. Having declared the attributes, they need to be initialized by the class constructor:



```

dynamicRefineFvMesh(io),
motionCoeffs_
(
    IOdictionary
    (
        IOobject
        (
            "dynamicMeshDict",
            io.time().constant(),
            *this,
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE,
            false
        )
    ).subDict(typeName + "Coeffs")
),
SBMFPtr_(solidBodyMotionFunction::New(motionCoeffs_, io.time())),
undisplacedPoints_
(
    IOobject
    (
        "points",
        io.time().constant(),
        meshSubDir,
        *this,
        IOobject::MUST_READ,
        IOobject::NO_WRITE,
        false
    )
)
)

```

In order to be able to use the added attributes to the new dynamic mesh class, their declarations need to be included in the header file:

```

#include "dictionary.H"
#include "pointIOField.H"
#include "solidBodyMotionFunction.H"

```

Like the `solidBodyMotionFvMesh`, the new class will rely on storing the new displaced mesh points as a private attribute and they will be calculated by the solid body motion function. The RTS of the solid body motion function is to be determined by the entries in `constant/dynamicMeshDict`. Since the new class inherits from `dynamicRefineMesh`, its constructor will expect to find the appropriate sub-dictionary in the `constant/dynamicFvMesh` file. Namely it is the `dynamicRefineFvMesh`-

Coeffs sub-dictionary that is used to set the parameters for refinement. As the new solid body mesh motion functionality is added to the class by composing the solid body motion function, the parameters for the function need to be defined somewhere. The function name used in the RTS process as well as a sub-dictionary that is used by the constructor of the motion function to set the function attributes need to be defined. The same approach that is used by other dynamic mesh classes is employed for the new class: within the **dynamicSolidBodyMotionRefineMesh-Coeffs** sub-dictionary, the parameters for the solid body motion are provided:

```
dynamicSolidBodyMotionRefinedFvMeshCoeffs
{
    solidBodyMotionFunction linearMotion;

    linearMotionCoeffs
    {
        velocity (0 0 -0.5);
    }
}
```

With these setting stored in **constant/dynamicFvMeshDict** file, read by the **movingRefinedMesh** a linear motion with the velocity of -0.5 m s^{-1} in the direction of the z axis is selected. Once the object of the new dynamic mesh class is constructed, the points of the mesh can be moved. This is done by extracting the relevant parts of the **dynamicSolidBodyMotionFvMesh::update** member function and placing them in the **update** member function the new class:

```
dynamicRefineFvMesh::update();

undisplacedPoints_ = this->points();

static bool hasWarned = false;

fvMesh::movePoints
(
    transform
    (
        SBMFPtr().transformation(),
        undisplacedPoints_
    )
);

if (foundObject<volVectorField>("U"))
{
```




```

        const_cast<volVectorField&>(lookupObject<volVectorField>("U"))
            .correctBoundaryConditions();
    }
    else if (!hasWarned)
    {
        hasWarned = true;
        WarningIn("solidBodyPointMotionSolver::update()")
            << "Did not find volVectorField U."
            << " Not updating U boundary conditions." << endl;
    }

    return true;

```

The second line in the `dynamicSolidBodyMotionRefinedFvMesh::update` member function ensures that the motion points are synchronized with the new mesh points generated after mesh refinement. This is absolutely necessary, as the point displacement only makes sense on point lists of the same length. The rest of the function shows the dependency of the dynamic mesh classes to the parent `fvMesh` class for the point motion. All boundary conditions of the velocity field need to be updated for mesh motion after the points of the mesh are moved. The declaration and definition for the `transform` function used to transform the points is still missing. The `transform` is a generic algorithm, so including the appropriate `.H` file is sufficient for using it. In order to use any fields in the code, the `volFields.H` file has to be included:

```

#include "transformField.H"
#include "volFields.H"

```

The library can be compiled and tested at this point.

If the parameters defined for the test case are used, when the case is run, overshoots in the `alpha1` field will appear, which can be seen in the log file when running

```
?> interDyMFoam 2>&1 | tee log
```

having first compiled the library with the changes described above.

```

Interface Courant Number mean: 0.0225 max: 0.6
Courant Number mean: 0.3375 max: 0.6
Time = 0.035

...

```

```
Execution time for mesh.update() = 0.18 s
MULES: Solving for alpha1
Phase-1 volume fraction = 0.183164 Min(alpha1) = 0 Max(alpha1) = 1.05625
MULES: Solving for alpha1
Phase-1 volume fraction = 0.185039 Min(alpha1) = 0 Max(alpha1) = 1.11145
MULES: Solving for alpha1
Phase-1 volume fraction = 0.186914 Min(alpha1) = 0 Max(alpha1) = 1.16561
MULES: Solving for alpha1
Phase-1 volume fraction = 0.188789 Min(alpha1) = 0 Max(alpha1) = 1.21875
```

In order to describe why overshoots in the field `alpha1` happen for the `interDyMFoam` solver, the numerical method behind the solution of the advection equation for this field should be described. However, describing all details of the algebraic two-phase advection algorithm for sharp fields is outside of the scope for this book. Still, users of the new dynamic mesh class expect solutions of field equations to work properly. If you haven't already read it, an introductory level background information of the numerical method used by OpenFOAM is provided in chapter 1. Now would be a good time to introduce yourself to this topic.

The fields are defined as functions of spatial coordinates and time. In the following, the generic symbol ϕ denotes a field:⁵

$$\phi = \phi(\mathbf{x}, t), \quad (13.3)$$

Equation 13.3 represents a field defined as a function of stationary spatial coordinates, in an inertial frame of reference. The mathematical models that are implemented by OpenFOAM solvers all assume this definition of a physical field. If a user of the solver wants to describe the change of the field in a relative (moving) frame of reference, there are ample options for that as well in OpenFOAM. Topological changes could be applied that add the layers of cells above the mesh, and remove the bottom layers for the simulation case with the motion similar to one of the `movingMeshRefined` case (e.g. a bubble rising in quiescent fluid). Alternatively, the mathematical model could be modified to take into account that the simulation is taking place in a moving frame of reference. This can be achieved in the solver directly when larger algorithmic changes are required, or using function objects to define explicit sources in the momentum equation. None of those methods are used for this example,

⁵This is very far from a strict mathematical definition, but it is more than sufficient for the purpose of explaining unbounded α_1 values.



though. The fields are expected to be modified using the volumetric flux field, as described in chapter 1.

The new dynamic mesh class in the way it is described up to this point, does not take any explicit care of modifying the volumetric flux fields, this is done within the `interDyMFoam` solver:

```
{
    // Calculate the relative velocity used to map the relative flux phi
    volVectorField Urel("Urel", U);

    if (mesh.moving())
    {
        Urel -= fvc::reconstruct(fvc::meshPhi(U));
    }

    // Do any mesh changes
    mesh.update();
}
```

Here if the mesh moving flag is set to true, a relative velocity field `Urel` will be *reconstructed* from the flux given by mesh motion. When `mesh.update()` is called, mesh refinement functions will use the refinement information (`mapPolyMesh`) to map the surface fields: update the surface field values for newly created faces created by splitting or merging old cell faces. The fields involved in the mapping are listed in the `constant/dynamicMeshDict`:

```
// Flux field and corresponding velocity field. Fluxes on changed
// faces get recalculated by interpolating the velocity. Use 'none'
// on surfaceScalarFields that do not need to be reinterpolated.
correctFluxes
(
    (phi none)
    (phiAbs none)
    (phiAbs_0 none)
    (nHatf none)
    (rho*phi none)
    (ghf none)
    (phi_0 none)
);
```

What can usually be observed for the volumetric flux field `phi` is, that it is mapped using the above computed velocity `Urel`. For this example, the field mapping procedure is disabled, since the mesh is being moved and refined at the same time. As the flux fields are not mapped against

the velocity fields, the flux overshoots are generated by the internal mapping procedure for the volumetric flux fields. Even if the flux correction is to be turned on in the above table, by providing names of cell centered velocity fields instead of the "none" entries, there would still be overshoots. When cell centered velocity fields are used to correct fluxes, they are interpolated at face centers, and the interpolation in the outer region of mesh refinement (cell layer next to unrefined cells) will suffer from higher non-orthogonality and aspect ratio errors. To solve this problem, the flux corrections done by the solver are avoided completely. Correcting the flux is done manually, by modifying the update member function of the new dynamic mesh class `dynamicSolidBodyMotionRefinedFvMesh` for the last time in this example:

```
surfaceScalarField& fieldPhi = const_cast<surfaceScalarField&>
(
    lookupObject<surfaceScalarField>("phi")
);
fieldPhi -= phi();

moving(false);
changing(false);
```

Again, it has to be stated that encapsulation was broken on purpose, by casting away constness of the volumetric flux field. However the update member function performs a similar action when mapping the fields - it is a direct consequence of the design strategy that registers fields to the mesh. Setting the moving and changing flags to false, any modification of the flux fields that might be performed by the solver are avoided. The line `fieldPhi -= phi();` basically states that the velocity coming from the solid body mesh motion is deducted from the velocity of the fluid. This allows relative motion of the fluid with respect to the mesh in a volume conservative way, but only for solid body motion which is divergence free. The solid body motion is divergence free, since no relative motion between body points is allowed and therefore there can be no decrease or increase in volume of any observed volume in the body. Compiling the library with the added new code for flux correction and executing the solver for the `movingRefinedMesh` test case shows no overshoots.

Figure 13.12 shows the simultaneously moved and refined mesh together with the `alpha1` field. The field stays exactly numerically bounded between 0 and 1, which can be checked by observing the simulation log



files in the example simulation case. The mesh is translated with a constant velocity downwards Figure 13.12 shows the mesh entering the view, and passing by the black fixed circle.

This development serves as an interesting example of extending the functionality of OpenFOAM, however it is still an educational example. It shows the relationship between the modified mesh and the fields, as well as the main guiding line in developing new mesh objects. There is much more to be understood when dynamic meshes are developed and applied on real world technical applications, but this is out of the scope of this book.

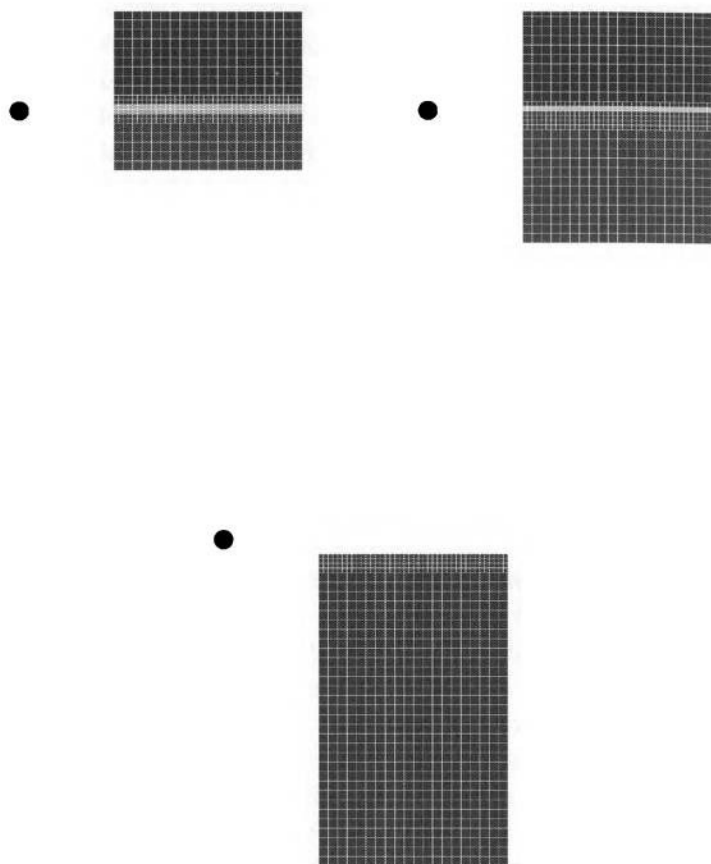


Figure 13.12: Three time step screenshots showing the `alpha1` field, the refined and moved mesh, and a circle fixed in the initial position of the interface. The mesh moves relative to the fluid interface while the interface stays fixed, and is also refined both locally and dynamically.

13.4 Summary

This chapter represents an attempt to introduce a powerful set of dynamic mesh operations available in OpenFOAM. The topic of dynamic mesh handling is probably wide and complicated enough to provide enough material for another book. An overview of the design of different dynamic mesh classes from the representative mesh handling categories is provided. Descriptions of the dynamic mesh usage are provided for selected dynamic mesh classes. Extension of a solver for dynamic mesh handling is described, as well as a description of the development of a new dynamic mesh class, involving combined mesh motion and hexahedral refinement. We hope that this chapter carries enough information within itself to form a solid starting point for those readers interested in using and extending dynamic mesh handling in OpenFOAM.

Further reading

- Bos, F. M. (2009). “Numerical simulations of flapping foil and wing aerodynamics”. PhD thesis. Technische Universiteit Delft.
- Bos, F. M., B. W. van Oudheusden, and H. Bijl (2013). “Radial basis function based mesh deformation applied to simulation of flow around flapping wings”. In: *Computers & Fluids*.
- Bos, F. M. et al. (2008). “OpenFOAM Mesh Motion using Radial Basis Function Interpolation”. In: *ECCOMAS*.
- Gamma, Erich et al. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2.
- Goldman, R. (2010). *Rethinking Quaternions: Theory and Computation*. Morgan & Claypool.
- Jasak (1996). “Error Analysis and Estimatio for the Finite Volume Method with Applications to Fluid Flows”. PhD thesis. Imperial College of Science.
- Jasak, H. and H. Rusche (2009). “Dynamic mesh handling in openfoam”. In: *Proceeding of the 47th Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, Orlando, Florida*.
- Jasak, H. and Z. Tukovic (2006). “Automatic mesh motion for the unstructured finite volume method”. In: *Transactions of FAMENA* 30.2, pp. 1–20.

- Juretić, F. (2004). “Error Analysis in Finite Volume CFD”. PhD thesis. Imperial College of Science.
- Menon, S. and D. P. Schmidt (2011). “Conservative interpolation on unstructured polyhedral meshes: an extension of the supermesh approach to cell-centered finite-volume variables”. In: *Computer Methods in Applied Mechanics and Engineering* 200.41, pp. 2797–2804.
- Menon, Sandeep (2011). “A numerical study of droplet formation and behavior using interface tracking methods”. PhD thesis. Graduate School of the University of Massachusetts Amherst.
- Menon, S. et al. (2008). “Simulating Non-Newtonian Droplet Formation With A Moving-Mesh Method”. In: *ILASS Americas, 21st Annual Conference on Liquid Atomization and Spray Systems*.



14

Outlook

The OpenFOAM platform is designed as a combination of a large number of libraries and applications that implement advanced methods in CFD. Addressing all aspects of the OpenFOAM platform, the way they interact with the rest of OpenFOAM and the many ways they can be extended following modern software development principles, is not possible to achieve in a single book.

Our goal when writing this book was to provide a solid background both for working with the platform as well as extending it, in a single document. For this purpose we have chosen those aspects of the platform that we have been in contact with, or those aspects that we were interested to learn more about. In this chapter brief notes are provided for those aspects of OpenFOAM that we didn't address, and might still be interesting topics in the future.

14.1 Numerical Methodology

Chapter 1 addresses the FVM with support for unstructured meshes (Weller, Tabor, Jasak, and Fureby 1998). However, in order to fully understand the numerical method in OpenFOAM, more details need to be provided. For example, discretization practices that are described in a multitude of doctoral dissertations based on OpenFOAM could be summarized and extended with numerical examples involving computational

domains with just a few cells. Such small numerical examples coupled with full descriptions of different operator discretization practices would fortify the knowledge on the FVM in OpenFOAM.

Another important topic when it comes to CFD are the algorithms used for the solution of Navier-Stokes equations (pressure-velocity coupling algorithms). There is a large number of segregated algorithms available and OpenFOAM is the right platform to be used for algorithmic design. It provides its own DSL for this purpose, often named *equation mimicking*. As an alternative to the segregated algorithm approach, an implementation of a block-coupled solution has been made available. Using and extending both the segregated and block-coupled algorithms, as well as their objective comparison when applied on compressible and incompressible flow simulations represent a topic of possible high interest.

Linear solver technology is a core part of the OpenFOAM framework, and we have not addressed this topic at all. Matrix storage format, the way it is related to the implementation of interpolation schemes, alternative formats, and linear solver algorithms, are all topics of importance for anyone that seeks to understand the details of a CFD method. Understanding the linear solver technology as well as the interpolation and discretization schemes represents a basis for extending the core of OpenFOAM to support alternative numerical methods and interfacing with other linear solver libraries.

14.2 Coupling to External Simulation Platforms

The modular object oriented and generic design of the OpenFOAM platform has lead to expansion of OpenFOAM into many different areas of physics simulation beyond iso-thermal fluid mechanics. In some aspects, some areas of numerics might be enhanced by coupling with other established libraries, which is also supported by the open source and modular nature of OpenFOAM.

One example of external platform coupling is the Fluid Structure Interaction (FSI), often modeled numerically as a coupled finite volume/finite element algorithm. In order to achieve the coupling between the methods, fluid stress conditions at a finite volume boundary might be communicated

to an external finite element stress analysis tool. The external platform then solves for the solid deformation magnitudes and the communication layer maps them back to the OpenFOAM case using dynamic mesh capabilities. Substantial work has been done in this direction by multiple researchers and describing the workflow for this family of problems within such a coupled computational framework would be beneficial.

14.3 Workflow Enhancement

14.3.1 Graphical User Interfaces

Many experienced OpenFOAM users and programmers prefer to work within the Linux environment for a variety of reasons. Unfortunately, a terminal interface can often be seen as intimidating and confusing to engineers considering the adoption of OpenFOAM as they are not used to working within the Unix/Linux operating system environment.

Recently, numerous GUI have been developed for OpenFOAM by commercial entities and some of them are freely available. One of the most popular GUI is HelyxOS, which can be obtained from engys.com. The graphical user interfaces often help setup configuration files used for mesh generation (`snappyHexMesh` meshing application) as well as configure solver dictionaries. In general, the GUIs can help new users get oriented with solver and case structure without requiring a strong background in Linux. An obvious advantage to using a GUI interface can be seen when configuring `snappyHexMesh` to easily work on multiple STL geometries. The ability to visualize, orient, and configure surface mesh resolutions on complex geometries can be taxing through the text interface.

One of the greatest attributes of OpenFOAM is its open source licensing and thus its ability to be freely customized. Unfortunately, this is often the greatest limitation in GUI designs. These interfaces are typically designed to create specific dictionaries for specific solvers in order to streamline the workflow. GUIs are most often not scanning and parsing the source code, instead they are built to work with known solver applications. Should a developer compile a new boundary condition or a transport model dictionary entry, the GUI would not be aware of such changes and would require development in order to include the new feature.

Describing the workflow with the available open source GUI frameworks, as well as how to further extend them might be a valuable future topic.

14.3.2 Console Interfaces

The advanced pyFoam¹ project that interfaces OpenFOAM with the Python programming language allows for advanced simulation run control, involving case parametrization, simulation monitoring, applications for streamlining the console workflow and much more. Documentation on pyFoam to a more detailed extent can be found on the OpenFOAM Wiki and related web pages. A more detailed description of various aspects of this important project, with examples of more complex applications is another interesting topic.

Another interesting project that combines the Python programming language and OpenFOAM is the pythonFlu² project, that focuses on adding interactivity to the OpenFOAM interface.

14.4 Unconstrained Mesh Motion

In this book, some mesh motion methods implemented in OpenFOAM were described. When a boundary is displaced too aggressively, however, the solution accuracy may degrade severely and the simulation might be destabilized all together. A few numerical methodologies exist outside of the boundary motion approach that can enable near unconstrained boundary motion or deformation. Examples of such methods are the immersed boundary method and the overset grid method.

14.4.1 Immersed Boundary Method

The immersed boundary method is a prominent method applied for unconstrained boundary motion. Here a boundary surface mesh is immersed in the background standard finite volume mesh. Using a local body force, the surface is accounted for by constraining the volumetric fluxes at the immersed boundary surface to be equal to zero. The finite volume mesh

¹See <http://openfoamwiki.net/index.php/Contrib/PyFoam> for more details.

²More information can be found at <http://pythonflu.wikidot.com/new-aboutproject>



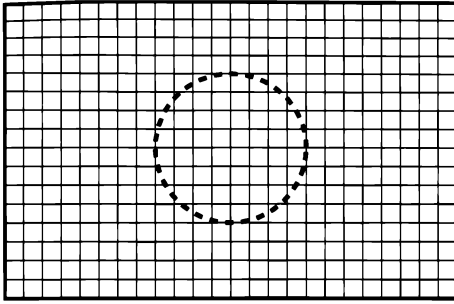


Figure 14.1: An example of the immersed boundary method applied to the flow over a cylinder. Thick line: Finite volume mesh boundary. Thick dashed line: Boundary mesh of the cylinder.

may incorporate a local refinement scheme to properly resolve small scale flow structures or stagnation points created from this patch. Mesh refinement can without issues reach the near vicinity of the interface as the mesh cells are not modified, contrary to the boundary fitted meshes. When the boundary fitted mesh is applied for domain discretization, often prismatic boundary cell layers are introduced to increase the accuracy near the boundary. The prismatic boundary layers cannot be refined easily with the existing dynamic refinement engine that supports hexahedral meshes.

A simplified illustration of the immersed boundary approach is shown in figure 14.1. Here, conditions near the immersed cylindrical boundary mesh are imposed such that the flow is influenced by its presence in the background mesh.

14.4.2 Overset Grid

In the overset grid method, two completely separate volume meshes are included in the simulation and interact with each other only via a subset of mesh cells in the overlapping region. Very accurate volume field mapping algorithm is required to properly conserve physical properties and efficient search algorithms are required to account for accurate relative mesh motion. An schematic image of the overset grid method is shown in figure 14.2. One advantage of this approach compared to immersed boundaries is that the secondary meshes can be constructed as boundary

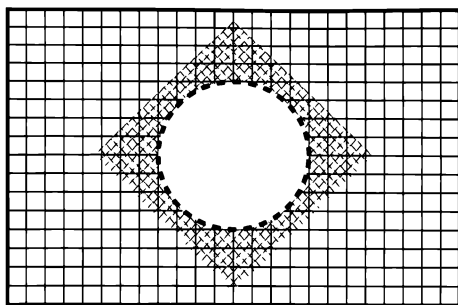


Figure 14.2: An example of overset grid method applied to the flow over a cylinder. Thick line: Finite volume mesh A boundary. Thick dashed line: Finite volume mesh B. Mesh A is continuous behind mesh B.

fitted meshes with boundary cell layers to increase the accuracy.

14.5 Summary

There are many useful and interesting topics in CFD that are being actively investigated by the wide OpenFOAM community. Already developed methodologies are either supported by the official release OpenFOAM release supported by ESI CFD, and some methodologies are integrated in the OpenFOAM-extend version. Due to the increasing user base, over time, more skilled developers using OpenFOAM technology across academia and industry will contribute to the overall capabilities of the code. More often then not, for both academic and industrial topics related to the official and extended version of the platform, the documentation describing the usage, design and possible extension of implemented methods remains to be a subject of lower priority.

There are many outlook topics that we might address in the future and there are topics that have been left out. Those topics that have not been addressed, were skipped not because of a lack of interest, but because of the focused scope of the book. We hope that this book contributes to an easier transfer of knowledge and will serve both as a source for learning as well as a reference to both the novice and experienced users and developers that work with the OpenFOAM technology.

Index

- S_f , 42
- 0-directory, 40, 93, 95
- 2D, 46, 80, 81, 253
- 3D, 80
- Advection, 25
- Allwclean, 198
- Allwmake, 198
- AMI, 35
- Application logic, 194
- Approximation, 16, 21, 26
- autoPatch, 81
- autoPatch, 83
- Bernoulli equation, 362
- bitbucket, 196
- blockMesh, 13, 40, 48--57, 85, 98,
124, 136, 237, 368, 398
 - arc, 50, 55
 - block order, 52
 - blocks, 50
 - convertToMeters, 52
 - coordinate system, 50
 - defaultFaces, 54
 - edgeGrading, 51
 - expansion ratio, 51
 - grading, 51
 - polyLine, 50
 - polyline, 55
 - simpleGrading, **51, 54**
 - spline, 50, 55
- blockMeshDict, 40, **48**
- Boundary condition, 95--100, 269,
388
 - Cauchy, 45
 - Dirichlet, 45
 - empty, 96
 - fixedValue, 192
 - fixedValue, 223
 - inletOutlet, 223
 - movingWall, 97
 - Neumann, 45
 - oscillatingFixedValue, 97
 - turbulentIntensityKineticEnergyInlet, 223
 - zeroGradient, 96
- Boundary layer thickness, 135
- Boundary mesh, 191
- boundaryField, 97
- C++11, 176
- CAD
 - STL, 47, 60
- calcTypes, 128
 - addSubtract, 128--129
 - components, 129
 - div, 129
 - interpolate, 129
 - mag, 129
 - magGrad, 129--130
 - magSqr, 129--130
 - randomise, 130
- Case
 - 0, *see* 0-directory
 - constant, *see* constant-directory
 - processor directories, *see* Processor directories
 - system, *see* system-directory
 - time directory, 82, 93
- Cell, 12, 16, 28, 32
- checkMesh, 53, 80, 84
- Class, 179, 192, 242

- class, 382
- Coefficient matrix, 161
- Collaboration, 195
- collapseEdges, 82
- Community Projects
 - funkySetFields, *see* funky-SetFields
 - swak4Foam, *see* swak4Foam
- Compiler, 179
 - Flags, 231--232
 - Optimization, 181
- compressibleInterFoam, 271
- Computational mesh, *see* Spatial discretization
- Conductivity coefficient, 15
- Constant reference, 192
- constant-directory, 40, 93--94, 278, 355, 396
 - dynamicMeshDict, 94
 - transportProperties, 93
 - turbulenceProperties, 93
- Constructor, 180, 189
 - Copy, 180, 189
 - Empty, 180
- Container, 209
- Control volume, *see* Cell
- controlDict, 98, 121--122, 233, 358, 360, 368, 372
 - outputControl, 361
 - outputTime, 361
 - purgeWrite, 122
 - writeControl, 121
- Convergence, 34
- copy elision, 176
- Copyright, 231
- Couette flow, 15, 334
- Courant number, 109, 261, 262
- curl, 161
- Cut-cell, 21
- Data conversion
 - foamMeshToFluent, *see* foamMeshToFluent
 - foamToStarMesh, *see* foamToStarMesh
- Data structure
 - volScalarField, 371, 375
- Data structures
 - autoPtr, 178--185, 244
 - controlDict, 360
 - dictionary, 159, 167--169
 - lookupOrDefault, 167
 - subDict, 168--169
 - toc, 167--168
 - dimensionedScalar, 172, 267, 274
 - dimensionedVector, 172
 - dimensionSet, 96, 169--173
 - DLList, 208
 - DynamicList, 208, 209
 - Field, 179
 - functionObject, 356--359
 - functionObjectList, 357
 - fvMesh, 154
 - fvVectorMatrix, 105, 175, 262
 - GeometricBoundaryField, 192
 - GeometricField, 189
 - HashSet, 355
 - HashTable, 209
 - Info, 202, 205
 - IOdictionary, 266
 - lookupOrDefault, 267
 - IOobject, 189, 266, 335
 - MUST_READ, 190
 - label, 41, 244, 355
 - labelHashSet, 355
 - labelList, 42
 - labelListList, 41
 - List, 209
 - pimpleControl, 265
 - regIOobject, 244, 355
 - scalar, 267
 - scalarField, 192, 244
 - surfaceVectorField, 129
 - Time, 160
 - Time, 357
 - timeSelector, 252
 - tmp, 156
 - tmp, 175, 178



- transportModel, 335
- triSurface, 245
- UList, 206
- vector, 119
- vectorField, 41
- volScalarField, 129, 274, 352
- volVectorField, 129
- word, 55, 244, 267, 415
- Debug Option, 180
- Debugging, 202, 207
 - gdb, 202--207
 - valgrind, 207--210
- DebugSwitches, 171--172
- decomposePar, 124
- decomposeParDict, 233
- decomposeParDict, 234
- delete, 176
- Design pattern, 244
- Destructor, 177, 180
- dimensions, 96
- div, 161
- Doxygen, 156, 197, 201, 230, 242
- dynamic polymorphism, 359
- dynamicMeshDict, 398
- Equation discretization, 25
- equation mimicking, 104, 163, 175
- Examples
 - bubbleCalc, 241
 - fallingDroplet2D, 360
 - isoSurfaceBubbleCalculator, 241
 - risingBubble2D, 136, 144, 241, 350, 355
 - unit cube, 52, 57
- Rising bubble 2D, 203
- explicit, 13, 27, 32, 105, 153, 161, 265, 387, 408
- Expression parser, 362
- extrudeMesh, 48
- extrudeMesh2D, 48
- Face centered, 191
- FatalError, 373
- Field
 - surfaceScalarField, 190
 - volScalarField, 189--190
- Fields, 188--192, 263--264
 - volScalarField, 188, 252
 - volTensorField, 188
 - volVectorField, 188
- Finite Volume Method, 14
- Floating Point Exception, 203
- foamCalc, 85, 128--130
- foamHelp, 97
- foamInstallationTest, 35
- foamJob, 121
- foamMeshToFluent, 79
- foamNew, 35, 178, 230
- foamNewCase, 35
- foamToStarMesh, 79
- foamToVTK, 355
- Function object, 93, 122, 165, 166, 262, 349--376, 392
- funkySetFields, 12, 229
- fvc, 105, 161, 162, 203
- fvm, 105, 161, 162
- fvSchemes, 104--116, 129, 160, 263
 - ddtSchemes, 108--110
 - backward, 108--110
 - bounded, 108
 - CoEuler, 108--109
 - CrankNicolson, 108
 - Euler, 108
 - localEuler, 108
 - SLTS, 108
 - steadyState, 108
 - divSchemes, 107, 114--115
 - gradSchemes, 110--114
 - cellLimited, 111--112
 - cellMDLimited, 111
 - edgeCellsLeastSquares, 111
 - faceLimited, 111
 - faceMDLimited, 111
 - fourth, 111
 - Gauss, 111
 - leastSquares, 111
 - pointCellsLeastSquares, 111, 113--114

- interpolationSchemes, 116, 161
- laplacianSchemes, 115
- fvSolution, 117--120, 167, 263, 265, 398
- nNonOrthogonalCorrectors, 119
- PISO, 117
- pRefCell, 119
- pRefPoint, 119
- pRefValue, 119
- solvers, 117
 - maxIter, 119
 - relTol, 118
 - tolerance, 118
- Garbage collection, 176
- Gauss divergence theorem, 26, 115, 161
- Gauss elimination, 33
- gcc, 178
- gcc, 181
- gdb, 202, 207
- Git, 210--213
- github, 196, 211
- gnuplot, 121, 133
- grad, 161
- Harmonic mean, 203
- heap-allocated, 176
- Heat transfer, 15
- HPC, 193, 213--234
- icoFoam, 98, 104, 266, 369
- IDE, 155
- implicit, 27, 32, 105, 115, 153, 161
- Info, 180
- Inheritance, 246
- Initial condition, 100--103
- interDyMFoam, 413
- interDyMFoam, 260
- interFoam, 144, 259, 267, 271, 273, 338, 361, 373
- internalField, 96
- Interpolation, 19, 21, 28, 113, 116
- ipython, 254
- Iso-surface, 241
- kinematic viscosity, 333
- Laplace Equation, 386
- Laplace's equation, 11
- laplacian, 161
- Linker, 194
- LTSInterFoam, 260
- Make
 - files, 199, 230, 251, 273, 368, 412
 - options, 181, 199, 231, 251, 261, 412
- makeAxialMesh, 81
- makeAxialMesh, 82
- Mathematical model, 11, 15, 104, 117
- Mathematical modeling, *see* Mathematical model
- Matrix, 32, 117
 - block, 153
 - coefficient, 32
 - preconditioning
 - DIC, 117
 - DILU, 117
 - solving, 33
 - direct, 33, 117
 - GAMG, 34
 - indirect, 117
 - iterative, 33
 - PBiCG, 117
 - PCG, 34, 117
 - sparse, 33, 117
 - under-relaxation, 34
- Mesh, 154
 - addressing, 22
 - axisymmetric, 81
 - block structured, 20
 - boundary, 40, 43, 82
 - boundary addressing, 24
 - Cell centre, 16, 41



- edge, 49
- Face area normal vector, 26, 28
- faces, 40, **41**
- indirect addressing, 22
- neighbour, 40, **43**
- nFaces, 82, 83
- owner, 40, **42**
- owner-neighbor addressing, 28
- owner-neighbour addressing, 24, 43
- points, 40, **41**
- polyMesh, 40, 41, 48, 60, 63, 80
- Refinement, 19
- scaling, 80
- structured, 17
- surface mesh, 39
- unstructured, 17
- vertice, 49
- volume mesh, 39
- Mesh generation, 12
 - blockMesh, *see* blockMesh, *see* blockMesh
 - blockMeshDict, *see* blockMesh-Dict
 - cfMesh, 68
 - enGrid, 67
 - extrude2DMesh, *see* extrude2DMesh
 - extrudeMesh, *see* extrudeMesh
 - refineHexMesh, *see* refine-HexMesh
 - snappyHexMesh, *see* snappy-HexMesh, *see* snappyHexMesh
- Mesh Motion, 381--391
- Meshing, *see* Mesh generation
 - Utilities, 85
- Momentum equation, 105, 175, 260
- mpirun, 124, 234
- Multiphase, 19, 259
- NACA 0012, 133, 235--240
- namespace, 106, 161, 203
 - calcTypes, *see* calcTypes
 - fvc, 203
 - fvm, 203
- Navier-Stokes Equation, 92, 220, 271
- new, 176
- non-constant reference, 174
- Object registry, 269
- Octree, 21
- OpenMPI, 217
- Operator
 - () , 351--352, 356
 - ++, 355, 370
 - ==, 354
 - [], 190
 - Assignment, 180
 - Overloading, 270
- Opt Option, 180
- Owner-neighbour addressing, 22, 209
- paraFoam, *see* paraview
- Parallel execution, 122--125, 232--234
 - MPI, 124
 - numberOfSubdomains, 123
 - scotch, 123
 - simple, 123
- Parallelization, 191
- paraview, 13, 98, 136, 143--149
 - Filters, 143
- pass-by-reference, 174
- pass-by-value, 174
- Patch, 45
 - cyclic, **46**
 - empty, **46**
 - patch, **45**
 - symmetryPlane, **46**
 - wall, **45**, 131
 - wedge, **46**, 81, 82
- patchAverage, 98, 131--132
- patchIntegrate, 131--132
- PDE, 15, 16, 32, 153, 157, 271--280
- Post-processing, 13, 127--135, 165--166, 225--226, 229--256
 - gnuplot, *see* gnuplot
 - paraview, *see* paraview

- patchAverage, *see* patchAverage
- patchIntegrate, *see* patchIntegrate
- pyFoamPlotWatcher, *see* pyFoamPlotWatcher
- sample, *see* sample
- Potential flow, 11, 40
- Pre-processing, 12, 224--225, 229--256
- Pressure-Velocity coupling, 117, 119--120, 153
 - PIMPLE, 118, 265
 - PISO, 117, 262
 - SIMPLE, 118
- probeLocations, 132--133
- Problem definition, 11
- Processor directories, 124
- Profiling, 207
- PyFoam, 232, 235--240
- pyFoamPlotWatcher, 121
- Python, 232, 235--240
 - numpy, 239
- python, 133
- PythonFlu, 237
- quaternion, 382
- Raw pointer, 177, 246
- reconstructPar, 125
- refineHexMesh, 85, 86
- Refinement, 12
- Repository, 197
- Residual, *see* Residuum
- Residuum, 33
- return-by-value, *see* pass-by-value, 181
- Reynolds number, 133
- Reynolds stress, 220
- Runtime selection, 159, 162
- sample, 135--143
 - fields, 136
 - setFormat, 136
- Scalar transport equation, 15, 271
- scalarTransportFoam, 400--404
- Segmentation Fault, 203
- septonion, 382
- setFields, 12, 85, 94, 102, 136, 280
- Sets
 - cellSet, 47
 - faceSet, 47
 - pointSet, 47
- Smart pointers, 174--188, 247
- snappyHexMesh, 13, 47, 48, 57--67
 - addLayers, 58
 - addLayersControls, 66
 - boundary layer, 66
 - castellatedMesh, 57, 62
 - Cell level, 60
 - expansionRatio, 67
 - finalLayerThickness, 67
 - geometry, 60, 61
 - Geometry definition, 60
 - locationInMesh, 65
 - minThickness, 67
 - mode, 64
 - Refinement, 63
 - refinementRegions, 64
 - refinementSurfaces, 63
 - relativeSizes, 67
 - searchableBox, 61
 - searchableSurfaceCollection, 62
 - snap, 57
 - snapControls, 65
 - triSurface, 61
 - triSurfaceMesh, 61
- snappyHexMeshDict, 58
- Solver, 35
 - icoFoam, 93
 - icoFoam, 79, 81, 92, 95, 117, 120, 123
 - interFoam, 85, 101, 136, 175
 - potentialFoam, 40
- Solving, 13
- Source term, 25, 163
- Spatial discretization, 12, 16
- Standard Template Library, 208



- swak4Foam, 12
- swak4foam, 229, 360, 362--365
- System of algebraic equations, *see* Matrix
- system-directory, 40, 94
 - controlDict, *see* controlDict, 94
 - fvSchemes, *see* fvSchemes
 - fvSolution, *see* fvSolution
- Taylor expansion, 113
- Temperature, 15
- Template, 179, 180, 192
- Turbulence, 11, 130, 219--226
 - y^+ , 130
 - DNS, 221
 - $k - \epsilon$, 220
 - $k - \omega$, 220
 - $k - \omega - SST$, 220
 - LES, 220
 - mixing length, 223
 - RANS, 220
 - specific dissipation rate, 223
 - turbulence intensity, 223
 - turbulent kinetic energy, 223
 - wall function, 46, 221--222
 - wallShearStress, *see* wall-ShearStress
- log-law, 222
- Tutorial
 - cavity, 120
- Tutorials, 35
 - cavity, 81, 92, 93, 95, 100, 117, 123, 136
 - damBreak, 85, 101
 - elbow, 79
 - pitzDaily, 40, 44
- type lifting, 354
- typedef, 180
- UML, 156, 201
- Utilities, 35, 85
 - autoPatch, 1
- blockMesh, *see* blockMesh, *see* blockMesh
- checkMesh, *see* checkMesh
- collapseEdges, *see* collapseEdges
- decomposePar, *see* decomposePar
- foamCalc, *see* foamCalc
- foamHelp, *see* foamHelp
- foamInstallationTest, *see* foamInstallationTest
- foamJob, *see* foamJob
- foamNew, *see* foamNew
- foamNewCase, *see* foamNewCase
- makeAxialMesh, *see* makeAxialMesh
- patchAverage, *see* patchAverage
- probeLocations, *see* probeLocations
- reconstructPar, *see* reconstructPar
- refineHexMesh, *see* refineHexMesh
- sample, *see* sample
- setFields, *see* setFields, *see* setFields
- snappyHexMesh, *see* snappyHexMesh, *see* snappyHexMesh
- vorticity, *see* vorticity
- wallShearStress, *see* wall-ShearStress
- wmake, *see* wmake
- yPlusLES, *see* yPlusLES
- yPlusRAS, *see* yPlusRAS
- VCS, 193
- virtual inheritance, 408
- Volume-of-Fluid, 260
- Volumetric flux, 93
- vorticity, 132
- VTK, 245
- wallShearStress, 135
- wclean, 182
- wclean, 276, 400

wmake

wmake, 371

wmake, 36, 197, 199, 215, 216, 230,
 231, 274, 276, 368, 403,
 412

wmakeLnInclude, 198

yPlusLES, 130--131, 225

yPlusRAS, 130--131, 134, 225

Zone, 47



The use of the OpenFOAM® toolkit for Computational Fluid Dynamics (CFD) is widely spread across industrial and academic environments. Compared to using proprietary CFD codes, the advantage of using OpenFOAM® lies in the Open-Source General Public License (GPL) licensing which allows the user to freely use and freely modify a modern high-end CFD code.

This book covers two main aspects of working with OpenFOAM®: using the applications and developing and extending the library code.

In the first part of the book, we chose a few utilities and applications to describe the OpenFOAM® work flow. This information should provide a sufficient starting point for the reader, who can investigate his/her interests further by following the provided instructions in a similar way for another solver or application. We have also provided an often missed general overview of the interplay between different toolkit elements involved in a CFD simulation using OpenFOAM® in hope to produce a top-view perspective of this complex CFD software to the reader.

The second part of the book describes how one can program with the OpenFOAM® library in a sustainable way using often encountered programming examples. We have tried to present the numerical background and software design of the specific programming example, together with its solution, hoping that the deeper understanding of the example will prepare the reader for other future programming tasks he or she may encounter.

OpenFOAM® and OpenCFD® are registered trademarks of OpenCFD Limited, the producer OpenFOAM software. All registered trademarks are property of their respective owners. This offering is not approved or endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks. Jens Höpken, Tomislav Maric, Kyle Mooney and sourceflux are not associated to OpenCFD.



sourceflux

ISBN 978-3-00-046757-8

